Approved for public release; distribution is unlimited.

The Design and Implementation of A Functional Interface
For The Attribute-Based Multi-Lingual Database System

by

Sybil B. Mack
Lieutenant, United States Navy
B.S., Savannah State College, 1984

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

Robert B. McGhee, Chairman
Department of Computer Science

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified.	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (*If applicable*) 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS Program Element No. / Project No. / Task No. / Work Unit Accession Number

11. TITLE (*Include Security Classification*)
The Design and Implementation of A Functional Interface for the Attribute-Based Multi-Lingual Database System

12. PERSONAL AUTHOR(S) Sybil B. Mack

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*year, month, day*)	15. PAGE COUNT
Master's Thesis	From To	1992 March	100

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (*continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUBGROUP	Database Design; Database Implementation; Database Management Systems; Functional Interface, Multi-Lingual Database, DAPLEX, Attribute-Based Language

19. ABSTRACT (*continue on reverse if necessary and identify by block number*)

Traditionally, the design and implementation of a conventional database system begins with the selection of a data model, followed by the specification of a model-based data language. Hence, the database system is restricted to a single data model and a specifie data. Hence, the database system is restricted to a single data model and a specific data language. One alternative to this traditional approach to database-system development is the multi-lingual database system (MLDS). This alternative approach affords the user the ability to access and manage a large collection of databases, via several data models and their corresponding data languages, without the traditional limitations.

This thesis presents a methodolgy for suporting the Functional Data Model and the Data Language, DAPLEX, for MLDS. Specifically, we design an interface which translates DAPLEX data language calls into attribute-based data language (ABDL) requests. A description of the software engineering aspects of the implementation and an overview of the modules which comprise our functional/DAPLEX interface are provided.,

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS	Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof. David K. Hsaio	22b. TELEPHONE (*Include Area code*) (408) 646-2253	22c. OFFICE SYMBOL Code 52Hq

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE: UNCLASSIFIED

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

THE DESIGN AND IMPLEMENTATION OF A
FUNCTIONAL INTERFACE FOR THE
ATTRIBUTE-BASED MULTI-LINGUAL
DATABASE SYSTEM

by

Sybil B. Mack
March, 1992

Thesis Advisor: David K. Hsiao

Approved for public release; distribution is unlimited

ABSTRACT

Traditionally, the design and implementation of a conventional database system begins with the selection of a data model, followed by the specification of a model-based data language. Hence, the database system is restricted to a single data model and a specific data language. One alternative to this traditional approach to database-system development is the multi-lingual database system (MLDS). This alternative approach affords the user the ability to access and manage a large collection of databases, via several data models and their corresponding data languages, without the aforementioned limitations.

This thesis presents a methodology for supporting the Functional Data Model and the Data Language DAPLEX for the MLDS. Specifically, we design an interface which translates DAPLEX data language calls into attribute-based data language (ABDL) requests. A description of the software engineering aspects of the implementation and an overview of the modules which comprise our DAPLEX language interface are provided.

c.1

TABLE OF CONTENTS

I. INTRODUCTION

A. MOTIVATION

During the past twenty years database systems have been designed and implemented using what we refer to as the traditional approach. The first step in the traditional approach involves choosing a data model. Candidate data models include the hierarchical data model, the relational data model, the network data model, the entity-relationship data model, or the attribute-based data model to name a few. The second step specifies a model-based data language, e.g., SQL for the relational data model, or DL/I for the hierarchical data model.

Numerous database systems have been developed utilizing this methodology. For example, in the sixties, IBM introduced the Information Management System (IMS), which supports the hierarchical data model and the hierarchical model-based data language, Data Language I (DL/I). In the early seventies, Sperry Univac introduced the DMS-1100, which supports the network data model and the network model-based data language, CODASYL Data Manipulation Language (CODASYL-DML). Recently, IBM introduced the SQL/Data System which supports the relational model and the relational model-based data language, Structured English Query Language (SQL). The result of this traditional approach to database system development is a homogeneous database system that restricts the user to a single data model and a specific model-based data language.

An unconventional approach to database system development, is referred to as the Multi-Lingual Database System (MLDS) [Ref. 1], alleviates the aforementioned restriction. This new system affords the user the ability to access and manage a large collection of databases via several data models and their corresponding data languages. The design goals of MLDS involve developing a system that is accessible via a hierarchical/DL/I interface, a relational/SQL interface, a network/CODASYL interface, and a functional/DAPLEX interface.

There are a number of advantages in developing such a system. Perhaps the most practical of these involves the reusability of database transactions developed on an existing database system. In MLDS, there is no need for the user to convert a transaction from one data language to another. MLDS permits the running of database transactions written in different data languages. Therefore, the user does not have to perform either a manual or an automated translation of existing transactions in order to execute a transaction in the MLDS. The MLDS provides the same results even if the data language of the transaction originates at a different database system.

A second advantage deals with the economy and effectiveness of hardware upgrade. Frequently, the hardware supporting the database system is upgraded because of technological advancements or system demands. With the traditional approach, this type of hardware upgrade has to be provided for all of the different database systems in use, so that all of the users may experience system performance improvements. This is not the case in MLDS, where only the upgrade of a single system, i.e., MLDS itself, is

necessary. In MLDS, the benefits of a hardware upgrade are uniformly distributed across all users, despite their use of different models and data languages.

A third advantage is that a multi-lingual database system allows users to explore the desirable features of the different data models and then use these to better support their applications. This is possible because MLDS supports a variety of databases structured in any of the well-known data models.

It is apparent that there exists ample motivation to develop a multi-lingual database system with many data model/data language interfaces. In this thesis, we are developing a functional/DAPLEX interface for MLDS. We are extending the work of Shipman [Ref. 4], who has shown the feasibility of this particular interface in a MLDS.

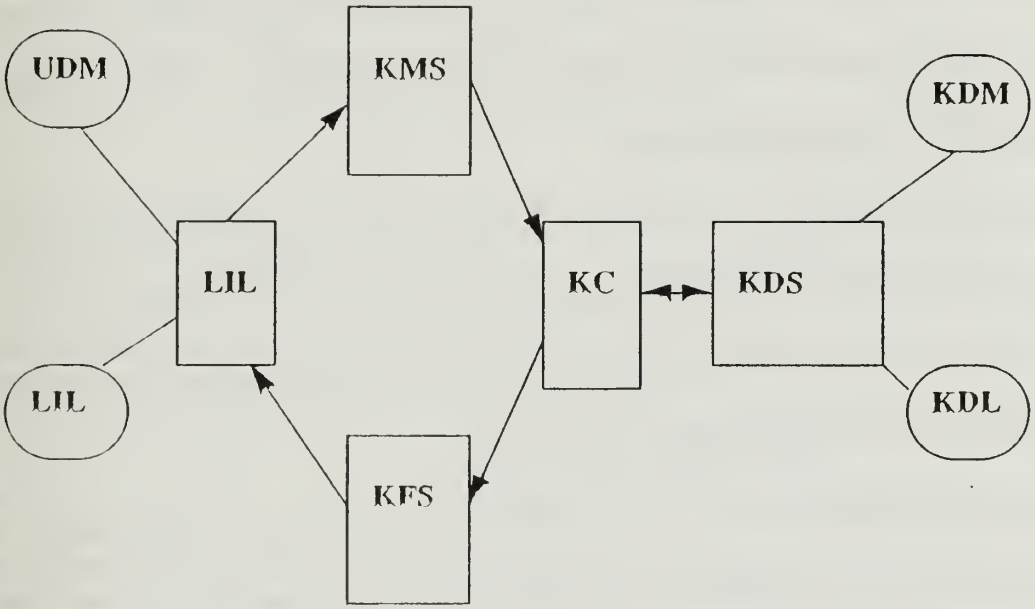
B. THE MULTI-LINGUAL DATABASE SYSTEM (MLDS)

A detailed discussion of each of the components of MLDS is provided in subsequent chapters. In this section we provide an overview of the organization of MLDS. This assists the reader in understanding how the different components of MLDS are related.

Figure 1 shows the system structure of a multi-lingual database system. The user interacts with the system through the language interface layer (LIL), using a chosen user data model (UDM) to issue transactions written in a corresponding model-based user data language (UDL). LIL routes the user transactions to the kernel mapping system (KMS). KMS performs one of two possible tasks. First, the KMS transforms an UDM-based database definition to a database definition of the kernel data model (KDM), when the user specifies that a new database is to be created. Second, when the user specifies that

an UDL transaction is to be executed, KMS translates the UDL transaction to a transaction in the kernel data language (KDL). In the first task, KMS forwards the KDM data definition to the kernel controller (KC). KC, in turn, sends the KDM database definition to the kernel database system (KDS). When KDS is finished with processing the KDM database definition, it informs KC. KC then notifies the user, via LIL, that the database definition has been processed and that loading of the database records may begin. In the second task, KMS sends the KDL transactions to the KC. When KC receives the KDL transactions, it forwards them to KDS for execution. Upon completion, KDS sends the results in the KDM form back to KC. KC routes the results to the kernel formatting system (KFS). KFS reformats the results from KDM form to UDM form. KFS then displays the results in the correct UDM form via LIL.

The four modules, LIL, KMS, KC, and KFS, are collectively known as the model language interface, for short, interface. Four similar modules are required for each of the model/language interface of MLDS. For example, there are four sets of these modules where one set is for the hierarchical/DL/I interface, relational/SQL interface, network/CODASYL interface, and the functional/DAPLEX interface. However, if the user writes the transaction in the native mode (i.e., in KDL), there is no need for an interface.



UDM : USER DATA MODEL
UDL: USER DATA LANGUAGE
LIL: LANGUAGE INTERFACE LAYER
KMS: KERNEL MAPPING SYSTEM
KC: KERNEL CONTROLLER
KFS: KERNEL FOMATTING SYSTEM
KDM: KERNELDATA MODEL
KDL: KERNEL DATA LANGUAGE
KDS: KERNEL DATABASE SYSTEM

Figure 1. The Multi-Lingual Database System

C. THE KERNEL DATA MODEL AND LANGUAGE

The choice of a kernel data model and a kernel data language is the key decision in the development of a multi-lingual database system. The overriding question, when making such a choice, is whether the kernel data model and kernel data language is capable of supporting the required data-model transformations and data-language translations for the language interfaces.

The attribute-based data model proposed by Hsiao [Ref. 5], extended by Wong [Ref. 6], and studied by Rothnie [Ref. 7], along with the attribute-based data language (ABDL), defined by Banerjee [Ref. 8], have been shown to be acceptable candidates for the kernel data model and kernel data language, respectively.

Why is the determination of a kernel data model and kernel data language so important for a MLDS? No matter how multi-lingual the MLDS may be, if the underlying kernel database system (i.e., KDS) is slow and inefficient, then the interfaces may be rendered useless and untimely. Hence, it is important that the kernel data model and kernel language be supported by a high-performance and great-capacity database system. Currently, only the attribute-based data model and the attribute-based data language are supported by such a system. This system is the multi-backend database (MBDS) [Ref. 1].

D. THE MULTI-BACKEND DATABASE SUPERCOMPUTER

The multi-backend database supercomputer (MBDS) has been designed to overcome the performance problems and upgrade issues related to the traditional approach of database computer design. This goal is realized through the utilization of multiple backends connected in a parallel fashion. These database backends have identical hardware, replicated software, and their own disk systems. In a multiple-backend configuration, there is a backend controller, which is responsible for supervising the execution of database transactions and for interfacing with the hosts and users. The backends perform the database operations with the database stored on the disk systems of the respective backends. The controller and backends are connected by a communication bus. Users access the system through either the hosts or the controller directly. MBDS is depicted in Figure 2.

Performance gains are realized by increasing the number of backends. If the size of the database and the size of the responses to the transactions remain constant, then MBDS produces a reciprocal decrease in the response times for the same user transactions when the number of backends is increased. On the other hand, if the number of backends is increased proportionally with the increase in databases and responses, then MBDS produces invariant response times for the same transactions. A more detailed discussion of MBDS is found in [Ref. 9].

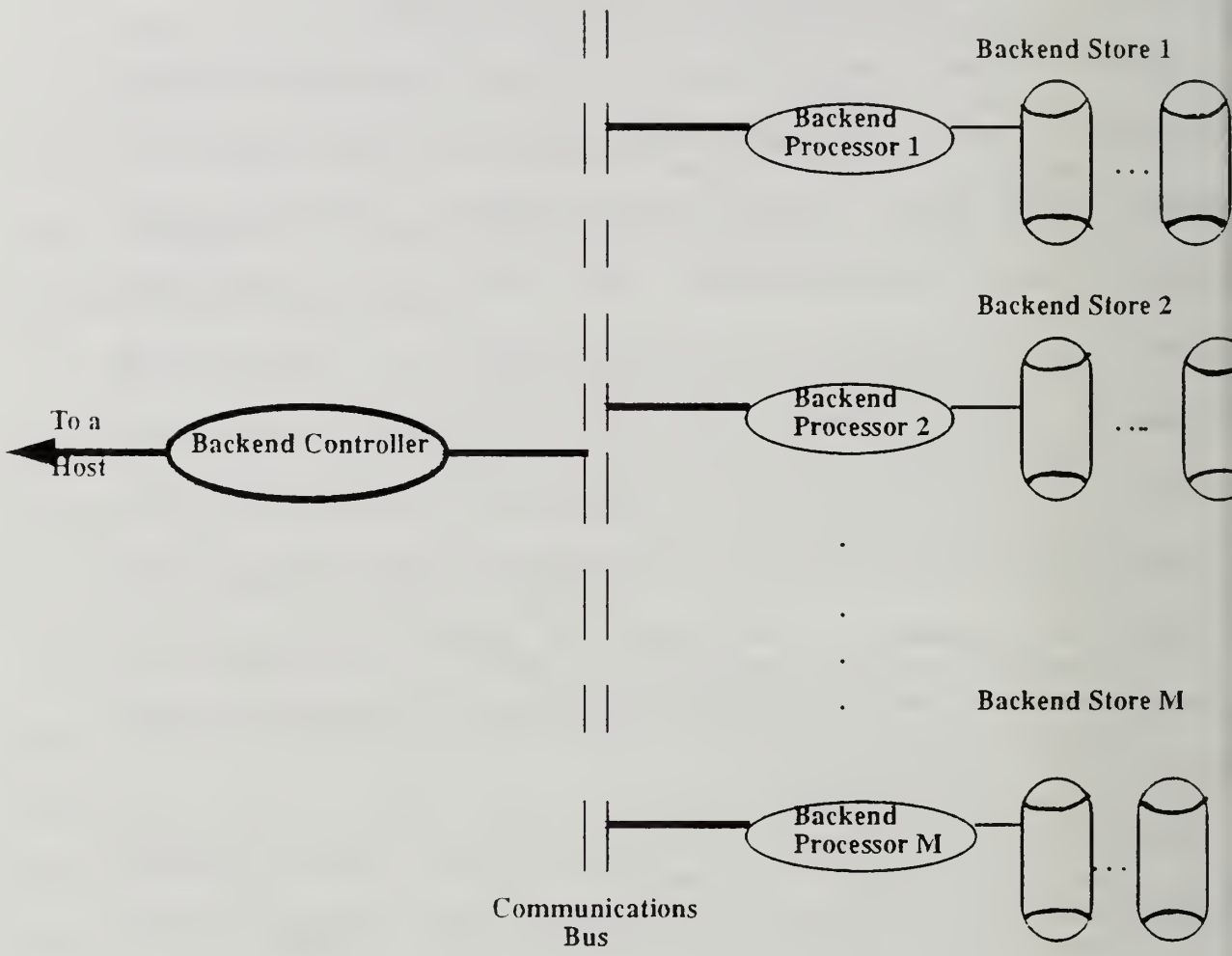


Figure 2. The Multi-Backend Database System.

In this thesis, we investigate the design of the functional-data-model and the functional-data language, i.e., (DAPLEX) interface for MLDS. Shipman [Ref. 4], provided the initial specification of DAPLEX. We are extending his specification to support the requirements of MLDS. In particular, we present a specification for the kernel mapping system (KMS) that will be used in the interface.

Throughout this thesis, we provide examples of DAPLEX requests and their translated ABDL equivalents. All examples involving database operations presented in this thesis are based on the education database described in Date [Ref. 10 pp. 644], as shown in Figure 3.

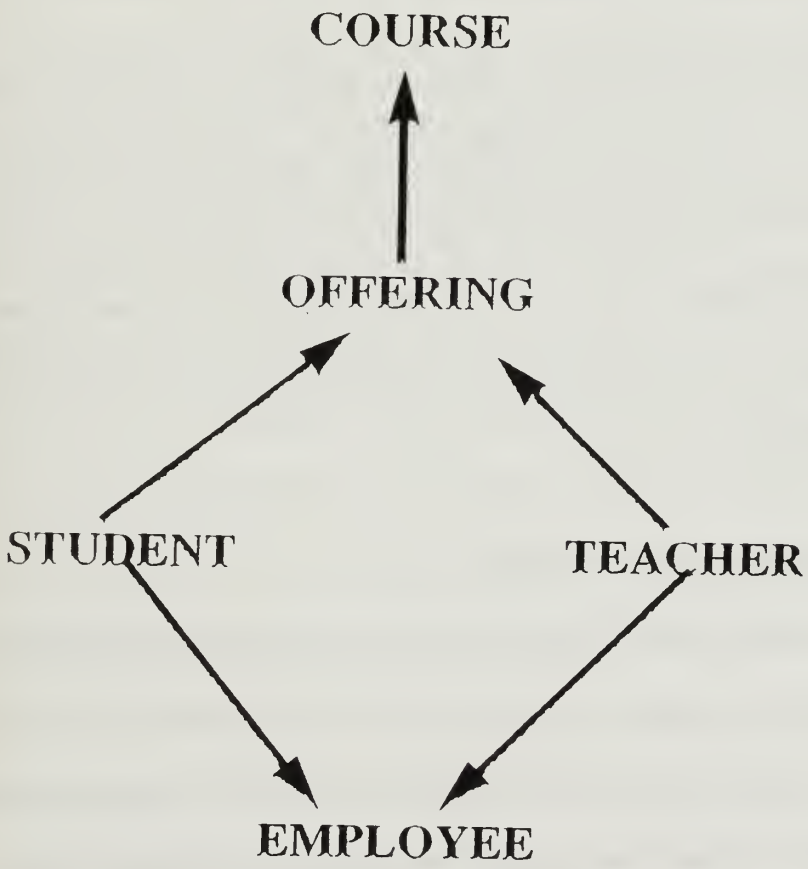


FIGURE 3. REFERENTIAL DIAGRAM FOR THE EDUCATION DATABASE

II. THE DATA MODELS

The choice of a kernel data model and a corresponding kernel data language is of vital importance in developing the multi-lingual database system (MLDS). The kernel data model and the kernel data language must be capable of supporting all the necessary data-model transformations and data-language translations required by the MLDS language interfaces.

It is our intention in this chapter to provide a description of the data models and languages related to the functional/DAPLEX interface, namely, the functional data model and the attribute-based data model and the DAPLEX data language and the attribute-based data language. A conceptual view of both models and languages is presented herein with a brief discussion of the data structures and language constructs associated with each model and language pair.

A. THE FUNCTIONAL DATA MODEL

The functional data model uses the concept of a mathematical function as its fundamental modeling construct. Any request for information can be visualized as a function call with certain arguments, and the function returns the required information. The main modeling primitives are entities and functional relationships among those entities. The functional data model and the functional language DAPLEX [Ref. 5] are selected for the functional interface of MLDS.

As defined in [Ref. 5], the basic concept of the functional data model consists of the entity and the function. These are intended to model conceptual objects and their properties. A function, in general, maps a given entity into a set of target entities. There are several standard types of entities at the most basic level; these are STRING, INTEGER, CHARACTER, REAL, etc. and are called printable entity types. Abstract entity types that correspond to real-world objects are given the type ENTITY as also described by Elmasri and Navathe [Ref. 6: pp. 444].

As discussed by Shipman [Ref 5: p. 141], some properties of an object are derived from properties of other objects to which it is related. For example, assume that each course has an "instructor of" property. We may then consider an "instructor of" property which relates students to their individual instructors. Such a property would be based on the "instructor of" property of those courses in which the student is enrolled. The principle of conceptual naturalness dictates that it be possible for users to treat such derived properties as if they were primitive. This follows, for example, even though the same real-world situation is being modeled, that properties which are "derived" in one database formulation may be "primitive" in another, and are termed derived functions.

The problem of database representation is complicated by the fact that no single model of reality may be appropriate for all users and problem domains. The properties which are considered relevant and the mechanisms by which they are most naturally referenced vary. Even the decision as to what constitutes an object depends on the real world is viewed assumed. To cope with these issues, the functional data model provides for the construction of separate user views of the database. Because user views are

specified in terms of derived functions, complex interrelationships among views may be accommodated.

Further, the functional data language DAPLEX, provides the user to specify the way the user models the problems. For example, consider the education database in Figure 3; we will show how some of the entity types and relationship types in that schema can be specified in a functional DAPLEX-like notation. The COURSE, OFFERING, STUDENT, TEACHER, EMPLOYEE entity types are declared in Figure 4.

```
COURSE() -> ENTITY
OFFERING() -> ENTITY
STUDENT() -> ENTITY
TEACHER() -> ENTITY
EMPLOYEE() -> ENTITY
```

Figure 4. A DAPLEX-like Specification

The statements in Figure 4 specify that the functions, COURSE, OFFERING, STUDENT, TEACHER, and EMPLOYEE return abstract entities, and hence these statements serve to define the corresponding entity types.

B. THE ATTRIBUTE-BASED DATA MODEL

The attribute-based data model was originally described by Hsiao [Ref.10]. It is a very simple but powerful data model capable of representing many other data models without loss of information. It is this simplicity and universality that makes the attribute-

based model the ideal choice as the kernel data model for MLDS, and the attribute-based data language (ABDL) as the kernel language for MLDS as well.

1. A Conceptual View of the Model

The attribute-based data model is based on the notions of attributes, and values for these attributes. An attribute and its associated value is therefore referred to as an attribute-value pair or keyword. These attribute-value pairs are formed from a Cartesian product of the attribute names and the domains of the values for the attributes. Using this approach, any logical concept can be represented by the attribute-based model.

A record, in the attribute-based model represents a logical concept. In order to specify the concept thoroughly, keywords must be formed. A record then, is simply a concatenation of the resultant keywords, such that no two keywords in the record have the same attribute. Additionally, the model allows for the inclusion of textual information, called the record body, in the form of a possibly empty string of characters describing the record or concept. The record body is not used for search purposes. Figure 5 gives the format of an attribute-based record:

```
(<attribute1,value1>, ...,
 <attributen,valuen>,
 { text })
```

Figure 5: An Attribute-Based Record

The angled brackets, <,>, are used to enclose a keyword where the attribute is first followed by a comma and then the value of the attribute. The record body is then set apart by curly brackets, {,}. The record itself is identified by the enclosure within parentheses. As can be seen from the above, this is quite a simple way of representing information.

In order to access the database, the attribute-based model employees an construct called predicates. A keyword predicate, or simply predicate is a triple of the form (attribute, relational operator, value). These predicates are then combined in disjunctive normal form to produce a query of the database. In order to satisfy a predicate, the attribute of a keyword in a record must be identical to the attribute in the predicate. Also, the relation specified by the relational operator of the predicate must hold between the value of the predicate, and the value of the keyword. A record satisfies a query if all predicates of the query are satisfied by certain keywords of the record. A query of two predicates as below

(TYPE = CSET) and (CNUM = K1)

would be satisfied by any record of **TYPE CSET** (course type) whose **CNUM** (course number) is **K1**, and it would have the following form,

(<attribute1,value1>, ...,<TYPE,CSET>, ... ,
<CNUM,K1>, ... ,<attributen,valuen>,{text}).

2. The Attribute-Based Data Language (ABDL)

ABDL as defined by Banerjee, Hsiao, and Kerr [Ref. 11] was originally developed for use with the Database Computer (DBC). This language is the kernel language used in MLDS. ABDL supports the five primary database operations, **INSERT, DELETE, UPDATE, RETRIEVE**, and **RETRIEVE-COMMON**. Those of importance to us in this portion of the MLDS work, however, are **INSERT, DELETE, UPDATE,** and **RETRIEVE**. A user of this language issues either a request or a transaction. A request in ABDL consists of a primary operation with a qualification. The qualification specifies the portion of the database that is to be operated on. When two or more requests are grouped together and executed sequentially, we have a transaction in ABDL. There are four types of requests, corresponding to the four primary database operations listed above. They are referred to by the same names.

A record is inserted into the database with an INSERT request. The qualification for this request is a list of keywords and a record body. Records are removed from the database by a DELETE request. The qualification for this request is a query.W h e n records in the database are to be modified, the UPDATE request is utilized. There are two parts to the qualification for this request. They are the query and modifier. The query specifies the records to be modified while the modifier specifies how the records are to be modified.

The final request to be mentioned here is the RETRIEVE request. As its name implies, it retrieves records from the database. The qualification for this request consists of a query, a target-list, and an optional by-clause. The query specifies the records to be retrieved. The target-list contains the output attributes whose values are required by the request, or it may contain an aggregate operation, i.e., **AVG, COUNT, SUM, MIN,** or **MAX**, on one or more output attribute values. The by-clause is optional and is used to group records when an aggregate operation is specified.

As indicated, ABDL consists of some very simple database operations. These operations, nevertheless, are capable of supporting complex and comprehensive transactions. Thus, ABDL meets the requirement of capturing all of the primary operations of a database system, and is quite useful for our purposes.

C. THE DATA STRUCTURES

The functional/DAPLEX language interface has been developed to provide two kinds of data: (1) data shared by all users, and (2) data specific to each user. The reader should realize that the data structures used in our interface, and described below, have been deliberately made generic. Hence, these same structures support not only our functional/DAPLEX interface, but the other model/language interfaces as well, i.e., relational/SQL, network/CODASYL-DML, and hierarchical/DL/I.

1. Data Shared by All Users

The data structures that are shared by all users, are the database schemas defined by the users thus far. In our case, these are functional schemas, consisting of segments

and attributes. These are not only shared by all the functional database users, but also shared by the four modules of MLDS, i.e., LIL, KMS, KC, and KFS. Figure 6 depicts the first data structure used to maintain data.

```
union  dbid_node
 {
     struct   rel_dbid_node   *rel;
     struct   hie_dbid_node   *hie;
     struct   net_dbid_node   *net;
     struct   ent_dbid_node   *ent;
 }
```

Figure 6. The dbid_node Data Structure.

It is important to note that this structure is represented as a union. Hence, it is generic in the sense that a user may utilize this structure to support other model/language interfaces. However, we concentrate only on the functional/DAPLEX interface. In this regard, the fourth field of this structure points to a record that contains information about an entity dbid node for a functional database. Figure 7 illustrates this record.

```
struct ent_dbid_node
 {
     char name[DBNLength + 1];
     struct     ent_non_node        *nonentptr;
     int        edn_num_nonent
     struct     ent_node           *entptr;
     int        edn_num_ent
     struct     gen_sub_node        *subptr;
     int        edn_num_gen
     struct     sub_non_node        *nonsubptr;
     int        edn_num_nonsub
     struct     der_non_node        *nonderptr;
     int        edn_num_der
     struct     overlap_node        *overptr;
     int        edn_num_ovr
     struct     ent_dbid_node       *next_db;
 }
```

Figure 7. The ent_dbid_node Data Structure.

The first field is simply a character array containing the name of the functional database. The next pointer field establishes each base-type nonentity node, followed by its integer value field of nonentity types. The next field pointer is for each entity node, followed by an integer value field of entity types. The next field defines for each generalization (supertype/subtype) nodes, followed by an integer value field of generalized subtypes. The next field is a pointer for each subtype nonentity node, followed by an integer value field of nonentity subtypes. The tenth field points to each derived type nonentity node, followed by an integer value field of nonentity derived types. The eleventh field points to the overlapping constraints in the database, followed by an integer value field of overlap nodes. The final field is simply a pointer to the next database.

Figure 8. is the structure definition for each base-type nonentity node. The first field of the record holds the name of the node. The next field serves as a flag to indicate the attribute type. For instance, an entity may either be an integer, a string, a floating point number, or boolean. The characters "i", "s", "f", "b" are used, respectively. The third field indicates the maximum length of base_type value. For example, if this field is set to ten and the type of this attribute is a string, then the maximum number of characters that a value of this attribute type may have is ten. The fourth field can be true or false depending on whether there is a range. If range exists, there must be two entries into ent_value. The fifth field indicates the number of actual values, and the six field indicates the actual value of base_type. The seventh field is boolean to reflect constant value. The last field is simply a pointer to the next ent_non_node.

```
struct ent_non_node
   {
     char      enn_name[ENLength  + 1 ];
     char      enn_type;
     int       enn_total_length;
     int       enn_range;
     int       enn_num_values;
     struct    ent_value          *enn_value
     int       enn_constant;
     struct    ent_non_node        *enn_next_node;
   }
```

Figure 8. The ent_non_node Data Structure.

Figure 9 is the structure definition for each entity node. The first field is an array which holds the name of the node. The second field keeps track of the unique id assigned to each entity type in the database. The third field indicates the number of associated functions. The fourth field if proven to be true (=1), indicates a terminal type. The fifth field is a function node pointer and the last field points to the next entity node.

```
struct ent_node
 {
     char        en_name[ENLength + 1 ];
     int         en_last_ent_id;
     int         en_num_funct;
     int         en_terminal;
     struct      function_node        *en_ftnptr;
     struct      ent_node             *en_next_ent;
 }
```

Figure 9. The ent_node Data Structure.

Figure 10 is the structure definition for each generalization (supertype/subtype) node. The first field is an array which holds the name of the node. The second field indicates the number of associated functions. The third field if proven to be true (=1), indicates a terminal type. The fourth field is a pointer to an entity supertype, whereas the fifth field provides the number of entity supertypes. The next field indicates a function node pointer, followed by the seventh field which indicates a pointer to a subtype supertype. The next field indicates the number of subtype supertypes present and the last field points to the next gen_sub node.

```
struct gen_sub_node

{
    char gsn_name[ENLength + 1 ];
    int         gsn_num_funct;
    int         gsn_terminal;
    struct      ent_node_list       *gsn_entptr;
    int         gsn_num_ent;
    struct      function_node       *gsn_ftnptr;
    struct      sub_node_list       *gsn_subptr;
    int         gsn_num_sub;
    struct      gen_sub_node        *gsn_next_genptr;

}
```

Figure 10. The gen_sub_node Data Structure.

Figure 11. is the structure definition for each subtype nonentity nodes. The first field of the record holds the name of the node. The next field serves as a flag to indicate the attribute type. For instance, an entity may either be an integer, a string, a floating point number, or boolean. The characters "i", "s", "f", "b" are used, respectively. The third field indicates the maximum length of subtype value. The fourth field can be true or false depending on whether there is a range. If range exists, there must be two entries into ent_value. The fifth field indicates the number of actual values, and the six field indicates the actual value of subtype. The last field is simply pointer to the next sub_non_node.

```
struct sub_non_node
 {
     char snn_name[ENLength + 1 ];
     char snn_type;
     int          snn_total_length;
     int          snn_range;
     struct       snn_num_values;
     struct       ent_value            *snn_value;
     struct       sub_non_node           *snn_next_node;

  }
```

Figure 11. The sub_non_node Data Structure.

Figure 12. is the structure definition for each derived type nonentity node. The first field of the record holds the name of the node. The next field serves as a flag to indicate the attribute type. For instance, an entity may either be an integer, a string, a floating point number, or boolean. The characters "i", "s", "f", "b" are used, respectively. The third field indicates the maximum length of subtype value. The fourth field can be true or false depending on whether there is a range. If range exists, there must be two entries into ent_value. The fifth field indicates the number of actual values, and the six field indicates the actual value of subtype. The last field is simply pointer to the next der_non_node.

```
struct der_non_node
 {
     char dnn_name[ENLength + 1 ];
     char dnn_type;
     int          dnn_total_length;
     int          dnn_range;
     struct       dnn_num_values;
     struct       ent_value        *dnn_value;
     struct       der_non_node     *dnn_next_node;
```

Figure 12. The der_non_node Data Structure.

Figure 13 is the structure definition for overlapping constraints. The first field of the record holds the name of the node. The second field indicates a pointer to subtype supertype. The third field indicates the number of subtype supertypes and the last field is simply a pointer to the next node.

```
struct overlap_node
 {
     char base_type_name[ENLength + 1 ];
     struct       sub_node_list        *snlptr;
     int          num_sub node;
     struct       overlap_node         *next;
 }
```

Figure 13. The overlap_node Data Structure.

2. Data Specific to Each User

This category of data represents information required to support each user's particular interface need. The data structures used to accomplish this need may be thought of as forming a hierarchy. At the root is the user_info record, shown in Figure 14, which maintains information on all current users of a particular language interface. The user_info record holds the ID of the user, a union that describes a particular interface, and a pointer to the next user.

```
struct  user_info
 {
     char        uid[UIDLength + 1];
     union       li_infoli_type;
     struct      user_info*next_user;
 }
```

Figure 14. The user_info Data Structure.

The union field is of particular interest to us. As noted earlier, a union serves as a generic data structure. In this case, the union may hold the data for a user accessing any one of the model/language interface via LIL. The li_info union is shown in Figure 15.

```
union  li_info
 {
      struct      sql_infoli_sql;
      struct      dli_infoli_dli;
      struct      dml_infoli_dml;
      struct      dap_infoli_dap;
 }
```

Figure 15. The li_info Data Structure.

We are only interested in the data structures containing user information that pertain to the functional/DAPLEX interface. This structure is referred to as dap_info and is depicted in Figure 16. The first field of this structure, curr_db, is itself a record and contains currency information on the database being accessed by a user. The second field, file, is also a record. The file record contains the file descriptor and file identifier of a file of DAPLEX transactions, i.e., either requests or database descriptions. The field, dml_tran, is also a record, and holds information that describes the DAPLEX transactions to be processed.

III. THE LANGUAGE INTERFACE LAYER (LIL)

LIL is the first module in the functional/DAPLEX mapping process, and is used to control the order in which the other modules are called. LIL allows the user to input transactions from either a file or the terminal. A transaction may take the form of either a database description (DBD) of a new database, or a DAPLEX request against an existing database. A transaction may contain multiple requests. This allows a group of requests that perform a single task, such as a looping construct in DAPLEX, to be executed together as a single transaction. The mapping process takes place when LIL sends a single transaction to KMS. After the transaction has been received by KMS, KC is called to process the transaction. Control always returns to LIL, where the user may close the session by exiting to the operating system.

LIL is menu-driven. When transactions are read from either a file or the terminal, they are stored in a data structure called dap_req_info. If the transactions are DBDs, they are sent to KMS the in sequential order. If the transactions are DAPLEX requests, the user is prompted by another menu to selectively choose an individual request to be processed. The menus provide an easy and efficient way for the user to view and select the methods of request processing desired. Each menu is tied to its predecessor, so that by exiting one menu the user is moved up the "menu tree." This allows the user to perform multiple tasks in one session.

A. THE LIL DATA STRUCTURES

LIL uses two data structures to store the user's transactions and control which transaction is to be sent to KMS. It is important to note that these data structures are shared by both LIL and KMS.

The data structure used by LIL is named dap_req_info and is shown in Figure 17. Each copy of this record represents a user transaction, and thus, is an element of the transaction list. The first field of this record, req, is a character string that contains the actual DAPLEX transaction. After all the lines of a transaction have been read, the line list is concatenated to form the actual transaction, req. The second field of this record, req_len, contains the length of this transaction. It is used to allocate the correct and minimal amount of the memory space for the transaction. The third field, in_req, is a pointer to a list of character arrays that each contains a single line of one transaction. If a transaction contains multiple requests, the fourth field, sub_req, points to the list of requests that make up the transaction. In this case, the field in_req is the first request of the transaction. The last field, next_req, is a pointer to the next transaction in the list of transaction.

```
char            *dri_req;
int         dri_req_len;
struct      temp_str_info *dri_in_req;
struct      dri_req_info  *dri_sub_req;
struct      dri_req_info  *dri_next_req;
```

Figure 17. The dap_req_info Data Structure.

B. FUNCTIONS AND PROCEDURES

LIL makes use of a number of functions and procedures in order to create the transaction list, pass elements of the list to KMS, and maintain the database schemas. We describe these functions and procedures briefly in the following sections.

1. Initialization

MLDS is designed to be able to accommodate multiple users, but is implemented to support only a single user. To facilitate the transition from a single-user system to a multiple-user system, each user possesses his or her own copy of the user data structure when entering the system. This user data structure stores all of the relevant data that the user may need during the session. All four modules of the language interface make use of this structure. The modules use many temporary storage variables, both to perform their individual tasks, and to maintain common data between modules. The transactions, in the user-data-language form, and the mapped-kernel-data-language form, are also stored in each user data structure. It is easy to see that the user structure provides consolidated, centralized control for each user of the system. When a user logs onto the system, a user data structure is allocated and initialized. The user ID becomes the distinguishing feature to locate and identify different users. The user data structures for all users are stored in a linked list. When new users enter the system, their user data structures are appended to the end of the list. In our current environment there is only

a single element on the user list. In a future environment, when there are multiple users, we simply expand the user list as described above.

2. Creating the Transaction List

There are two operations the user may perform. A user may define a new database or process DAPLEX requests against an existing database. The first menu that is displayed prompts the user to select the operation desired. Each operation represents a separate procedure to handle specific circumstances. The menu looks like the following:

Enter type of operation desired
(l) - load a new database
(p) - process existing database
(x) - return to the operating system
ACTION ----> _

For either choice (i.e., **l** or **p**), another menu is displayed to the user requesting the mode of input. This input may always come from a data file. If the operation selected from the previous menu had been **"p"**, then the user may also input transactions interactively from the terminal. The generic menu looks like the following:

Enter mode of input desired
(f) - read in a group of transactions from a file
(x) - return to the main menu

ACTION ----> _

Each mode of input selected corresponds to a different procedure to be performed. The transaction list is created by reading from the file or terminal, looking for an end-of-transaction marker or an end-of-file marker. These flags tell the system when one transaction has ended, and when the next transaction begins. When the list is being created, the pointers to the list are initialized. The pointers are then set to the first transaction read, in other words, the head of the transaction list.

3. Accessing the Transaction List

Since the transaction list stores both DBDs and DAPLEX requests, two different access methods have to be employed to send the two types of transactions to KMS. We discuss the two methods separately. In both cases, KMS accesses a single transaction from the transaction list. It does this by reading the transaction addressed by the request pointer. Therefore, it is the job of LIL to set this pointer to the appropriate transaction before calling KMS.

a. Sending DBDs to KMS

When the user specifies the filename of DBDs (input from a file only), any further user intervention is not required. To produce a new database, the transaction list of DBDs is sent to KMS via a program loop. This loop traverses the transaction list, calling KMS for each DBD in the list.

b. Sending DAPLEX Requests to KMS

In this case, after the user has specified the mode of input, the user conducts an interactive session with the system. First, all DAPLEX requests are listed

to the screen. As the requests are listed from the transaction list, a number is assigned to each transaction in ascending order, starting with the number one. The number appears on the screen to the left of the first line of each transaction. Note that each transaction may contain multiple requests. Next, an access menu is displayed which looks like the following:

```
Pick the number or letter of the action desired
     (num) - execute one of the preceding transactions
     (d)   - redisplay the list of transactions
     (r)   - reset the currency pointer to the root
     (x)   - return to the previous menu
ACTION ----> _
```

Since DAPLEX requests are independent items, the order in which they are processed does not matter. The user has the option of executing any number of DAPLEX requests. A loop causes the menu to be redisplayed after any DAPLEX request has been executed so that further choices may be made. The "r" selection causes the currency pointer to be repositioned to the root of the functional schema so that subsequent requests may access the complete database, rather than be limited to the beginning of a current position established by previous requests.

4. Calling KC

As mentioned previously, LIL acts as the control module for the entire system. When KMS has completed its mapping process, the transformed transactions must be sent to KC to interface with the kernel database system. For DBDs, KC is called after all

DBDs on the transaction list have been sent to KMS. The mapped DBDs have been placed in a mapped transaction list that KC is going to access. Since DAPLEX requests are independent items, the user should wait for the results from one DAPLEX request before issuing another. Therefore, after each DAPLEX request has been sent to KMS, KC is immediately called. The mapped DAPLEX requests are placed on a mapped transaction list, which KC may easily access.

5. Wrapping-Up

Before exiting the system, the user data structure described in Chapter II must be deallocated. The memory occupied by the user data structure is freed and returned to the operating system. Since all of the user structures reside in a list, the node for the exiting user must be removed from the list also.

```
struct  dap_info
 {
     struct      curr_db_info      curr_db;
     struct      file_info         file;
     struct      tran_info         dml_tran;
     struct      ddl_info          *ddl_files;
     int         operation;
     int         answer;
     int         error;
     int         buff_count;
     union       kms_info          kms_data;
     union       kfs_info          kfs_data;
     union       kc_info           kc_data;
 }
```

Figure 16. The dap_info Data Structure.

The next field, ddl_files, is a pointer to a record describing the descriptor and template files. These fields contain information about the ABDL schema corresponding to the current functional database being processed, i.e., the ABDL schema information for a newly defined functional database. The next field of the dap_info record, operation, is a flag that indicates the operation to be performed. This may be either the loading of a new database or the execution of a request against an existing database. The next field, answer, is used by LIL to record answers received through its interaction with the user of the interface. The next field, error, is an integer value representing a specific error type. The buff_count field is a counter variable used in KC to keep track of the result buffers. The following fields, kms_data, and kfs_data, are unions that contain information required by KMS and KFS.

IV. THE KERNEL MAPPING SYSTEM (KMS)

KMS is the second module in the DAPLEX mapping interface and is called from the language interface layer (LIL) when LIL has received DAPLEX requests input by the user. The function of KMS is to (1) parse the request to validate the user's DAPLEX syntax, (2) translate or map the request to equivalent ABDL request(s), and (3) perform a semantic analysis of the current ABDL request(s) generated during a previous call to KMS. Once an appropriate ABDL request or a set of requests has been formed, it is made available to the kernel controller (KC) which then prepares the request for execution by MBDS. KC is discussed in Chapter V.

A. AN OVERVIEW OF THE MAPPING PROCESS

From the description of the KMS functions above we immediately see the requirement for a parser as a part of KMS. This parser validates the DAPLEX syntax of the input request. The parser grammar is the driving force behind the entire mapping system.

1. The Parser / Translator

As the low-level input routine, we utilize a Lexical Analyzer Generator (LEX) [Ref. 16]. LEX is a program generator designed for lexical processing of character input streams. Given a regular-expression description of the input strings, LEX generates a

program that partitions the input stream into tokens and communicates these tokens to the parser.

The KMS parser has been constructed by utilizing Yet-Another-Compiler Compiler (YACC) [Ref.]. YACC is a program generator designed for syntactic processing of token input streams. Given a specification of the input language structure (a set of grammar rules), the user's code is to be invoked when such structures are recognized. YACC generates a program that syntactically recognizes the input language and allows invocation of the user's code throughout this recognition process. The class of specifications accepted is a very general one such as the LALR(1) grammars. It is important to note that the user's code mentioned above is our mapping code that is going to perform the DAPLEX-to-ABDL translation.

The parser produced by YACC consists of a finite-state automaton with a stack and performs a top-down parse, with left-to-right scan and one token look-ahead. Control of the parser begin initially with the highest-level grammar rule. Control descends through the grammar hierarchy, calling lower and lower-level grammar rules which search for appropriate tokens in the input. As the appropriate tokens are recognized, some portions of the mapping code may be invoked directly. In other cases, these tokens are propagated back up the grammar hierarchy until a higher-level has been satisfied, at which time further translation is accomplished. when all of the necessary lower-level grammar rules have been satisfied and control has ascended to the highest-level rule, the parsing and translation processes are complete.

2. The KMS Data Structures

KMS utilizes, for the most part, just five structures defined in the interface. It, naturally, requires access to the DAPLEX input request structure discussed in Chapter II, the dap_req_info structure. However, the five data structures to be discussed here are only those unique to KMS.

The first of these, shown in Figure 18, is a record that contains information accumulated by KMS during the grammar-driven parse that is not of immediate use. This record allows the information to be saved until a point in the parsing process where it may be utilized in the appropriate portion of the translation process. The first fields in this record, tempt_ptr, name1_ptr and id, are pointers to the head of the list of attribute and the initial name selected. This is the name of the attribute whose values are retrieved from the database. The third field, overfirst_ptr, is a pointer of the general sub_node list, the fourth field, der_non points to each derived type nonentity nodes. The next field is a pointer for each subtype nonentity node, followed by a pointer which establishes each base-type nonentity nodes. The next field is a pointer for each function type declaration, followed by the next field which indicates the actual value of base_type. The remaining structures are described in the remaining text.

```
struct  dap_kms_info
{
        struct      ident_list       *dki_temp_ptr;
        struct      ident_list       *dki_name1_ptr;
        struct      ident_list       *dki_id_ptr;
        struct      sub_node_list     *dki_overfirst_ptr;
        struct      der_non_node       *dki_der_non;
        struct      sub_non_node       *dki_sub_non;
        struct      ent_non_node       *dki_ent_non;
        struct      function_node     *dki_funct;
        struct      ent_value        *dki_ev_ptr;
        struct      dap_create_list    *dki_create;
        struct      req_line_list     *dki_req_ptr;
        struct      create_ent_list    *dki_cel_ptr;
        struct      overlap_node       *dki_create_ovrptr;
        struct      ent_value_list    *dki_evl_ptr;
        struct      dml_statement      *dml_statement_ptr;
        struct      loop_info         *loop_info_ptr;
}
```

Figure 18. The dap_kms_info Data Structure

Figure 19 is the structure for dap_create_list. It contains an integer field to hold the req_type from an insert or a retrieve request. The structures that support this list is the dap_av_pair_list and the dap_create_list. The dap_av_pair_list generally contains a single item, which points to a single insert list. However, in the case of multiple path insertion, the list contains an item that points to each insert list, corresponding to each entity type to be inserted. The create_list is a pointer to the next list to be inserted.

```
struct  dap_create_list

{
        int             req_type;
        char            en_name[ENLength + 1];
        struct          dap_av_pair_list     *av_pair_ptr;
        struct          dap_create_list      *next;
}
```

Figure 19. The dap_create_list Data Structure.

The dap_av_pair_list contains a character string to receive the name length to include a stru define the function node to determine the number of values to be inserted for a given av_pair also contains an entity value pointer to determine the entity values in the insert lists. The last the dap_av_pair_list pointer to the next list to be evaluated.

```
struct dap_av_pair_list
{
        char  name[ENLength + 1 ];
        struct function_node        *ftnptr;
                intnum_value
        struct ent_value              *valptr;
        struct dap_av_pair_list         *next;
};
```

Figure 20. The dap_av_pair_list Data Structure.

The req_line_list structure contains the REQLength constant for defining the maximum lengths of entity names, respectively. The last field is a pointer to the next entity in

```
struct req_line_list

    {
       char   req_line[REQLength];
       struct req_line_list      *next;
    }
```

Figure 21. The req_line_list Data Structure.

The create_ent_list contains two pointer structures, one for the entity nodes in the list and one or the subtypes with one or more entity supertypes. The last field is a pointer to the next node in the ist.

```
struct  create_ent_list

  {
    struct ent_node_list   *enl_ptr;
    struct sub_node_list   *snl_ptr;
    struct create_ent_list *next;

  }
```

Figure 22. The create_ent_list Data Structure.

The ent_value_list was discussed in Chapter II. The dml_statement is a pointer which points to any DAPLEX expression. This structure provides an integer for any type of expression hat is in the list such as Assignment, Include, Exclude, Destroy, Move, Procedure, and Create. The econd field is the expression pointer followed by the field to check the index to see if the component s indexed. This is followed by the basic expression list field to check for literary context. The last tructure compares the expression with associated expression in the list.

```
struct  dml_statement

 {
     int   type;
     struct      dap_expr_info*dap_expr_ptr;
     struct      indexed_component*indexed_comp_ptr;
     struct      basic_expr_list*basic_expr_ptr;
     struct      comp_assoc_list*comp_assoc_ptr;

}
```

Figure 23. The dml_statement Data Structure.

The loop_info structure is the last structure in kms_info. This structure contains a structure for the domain_info. This establishes the fact that only one of the first tw is active at one time. The next structure order_comp_list is a pointer that defines the sort orde list. It indicates if the order of the list is ascending or descending. The last field in the structur dml_statement2 pointer to the next statement list.

B. The Attribute-Based Data Language (ABDL)

The access and manipulation of a database are performed through five primary operations delete, update, retrieve and retrieve-common). Four of these operations are formed by utiliz queries as just described. A brief description of each operation follows.

The **INSERT** request is used to insert a new record into a specified file of an existing datab takes the form:

INSERT Record

An example of an **INSERT** operation which inserts a student record into a file named Student is:

INSERT(<FILE = Student>, <SNAME = Baker>,<SNUM = 8942>)

The **DELETE** operation is used to remove one or more records from the database. A **DELETE** operation takes the form:

DELETE Query

An example operation is used to remove one or more records from the database. A **DELETE** which removes all students named 'Jackson' from the Student file is:

DELETE ((FILE = Student) and (SNAME = Jackson))

An **UPDATE** is used to modify records of the database. An **UPDATE** request consists of two parts. The syntax is:

UPDATE Query Modifier

An example of an **UPDATE** request which changes the grade of a student named Margaret to an 'A' is:

UPDATE ((FILE = Student) and (SNAME = Oliver) (GRADE = 'A'))

A **RETRIEVE** request is used to retrieve records from the database. The database is not altered operation. A **RETRIEVE** consists of three parts, a query, a target-list and an optional by-claus target list specifies the set of attribute values to be output to the user. It may consist of an ag operation (avg, count, sum, min, or max). The by-clause is used to group the output record syntax for a **RETRIEVE** request is:

RETRIEVE (Query) (Target-list) (By-clause)

For example:

RETRIEVE (FILE = Student) (SNAME) BY SNUM

would retrieve the names of all students, ordered by their student number.

The final operation is the **RETRIEVE-COMMON** request. It is used to merge two common attribute values. The syntax for a **RETRIEVE-COMMON** request is:

RETRIEVE (Query 1) (Target-list 1)
COMMON (attribute 1, attribute 2)
RETRIEVE (Query 2) (Target-list 2)

An example of such a request is:

```
RETRIEVE (FILE = STUDENT) (SNAME)
COMMON (SNAME, TNAME)
RETRIEVE (FILE = TEACHER) (TNAME)
```

This request would display a list of students and teachers that share a common name. As with the retrieve command, the database is not modified by this operation.

C. The Data Manipulation Language (DAPLEX)

1. The Retrieve Query

Data retrieval is the most basic operation of DAPLEX. Mapping indicates that a known quantity (DEPARTMENT) = 'Math', is to be transformed into a desired quantity (LNAME) by means of an attribute (L). The entities to be returned are listed in the RETRIEVE clause. The WHERE clause specifies the retrieval conditions. As an example, if we desire to retrieve the last names of all students majoring in 'Math'; we could write the following DAPLEX query:

```
RETRIEVE LNAME(MAJORING_IN(DEPARTMENT))
WHERE DNAME(DEPARTMENT) = 'Math'
```

This is because the inverse function MAJORING_IN(DEPARTMENT) returns STUDENT entities, so we can apply the derived function LNAME(STUDENT) to those entities.

The **RETRIEVE** construct allows the user great flexibility in data retrieval operations. To declare composite attributes, such as NAME, we have to declare them to be entities and then declare their

component attributes as functions. There are many other possible variations to the RETRIEVE op including the extremely useful nested functional notation. In the nested RETRIEVE, the result **RETRIEVE** request is embedded in the declaration clause to which it is applied.

2. The Insert Query

The **INSERT** request allows the user to insert a new argument or attributes about a p Insertion of a single attribute can be accomplished through the use of a query. For example, suppose we want to insert the middle initial of a student majoring in Computer Science named M A. Tucker.

```
FOR A Student
BEGIN
      LET Name (Student) = "Michael Tucker"
      LET MINIT(Student) = "A"
      LET Dept (Student) = THE Department SUCH THAT
       Name (Department) = "Computer Science"
  INSERT MINIT(Name(Student))
PRINT Name(Student)
```

3. The Update Query

UPDATE statements are used to specify the value returned by a function when applied to particular entities. Some examples illustrate the syntax involved.

"Add a new student named Fred the Math department."

```
FOR A NEW STUDENT
 BEGIN
  LET Name (Student) = "Bill"
  LET Dept (Student) = THE Department SUCH THAT
  Name (Department) = "Math"
```

In order to enroll the above student in a particular course in the Math department such as Differential Equations; use the above syntax and add the following:

```
LET Course (Student) = THE Course SUCH THAT
    Name (Course)  = "Differential Equations"

END
```

The following illustrates the incremental updating of multi-valued functions. Drop 'Introductory Calculus' from Fred's courses and add 'Calculus II',"

```
FOR THE Student SUCH THAT Name (Student) = "Fred"
 BEGIN
 EXCLUDE Course (Student) =
     THE Course SUCH THAT Name (Course) =
       "Introductory Calculus"
 INCLUDE Course (Student) =
     THE Course SUCH THAT Name (Course) =
       "Calculus II"
END
```

UPDATE statements set the value that a function is to return when it is applied to particular arguments. In the context of a DAPLEX expression, however, a function's arguments are not always individual entities but rather sets of entities. This is simply a result of the fact that the argument to a function is an expression which in general evaluate to a set. When a function is evaluated, the result is the union of all entities

returned by the function applied to all members of its argument set. Thus, "List all courses taken by CS students,"

```
FOR EACH
   Course (Student SUCH THAT Dept (Student) = "CS")
  PRINT Title (Course)
```

The argument to the "Course" function here is a set of "Student" entities. The evaluation of the function returns the set of all courses taken by any of these students. Note that each course is listed only once. Shipman [Ref. 2, pp. 151].

4. The Delete Query

The **DELETE** specifies attributes to be removed from the database. The attributes are indicated by means of an expression value and the an expression role. The expression value is the set of entities returned by evaluating the expression. The expression role is the entity type under which these entities are to be interpreted when resolving external function name ambiguities. As an example, to **DELETE** Fred Forest from the course database for the math curriculum mentioned above, we may use the following query.

```
DELETE NAME(MAJORING_IN(DEPARTMENT))
WHERE DNAME(DEPARTMENT) = "Math" AND
NAME(PERSON) = "Fred Forest"
```

The Name 'Fred Forest' is deleted from the database for the math department, not from the university's database.

D. The DAPLEX-to-ABDL Mapping

When the user wishes LIL to process requests against an existing database, the first task of KMS is to map the user's DAPLEX requests to equivalent ABDL requests.

1. The DAPLEX RETRIEVE Calls to the ABDL RETRIEVE

KMS maps each DAPLEX **RETRIEVE** request into as in most cases, a series of ABDL **RETRIEVE** requests. An operator identification flag is set during the translation process which allows KC to associate the appropriate operation to these requests for controlling their execution.

If we take the retrieve query stated in section C:

RETRIEVE LNAME(MAJORING_IN(DEPARTMENT))
WHERE DNAME(DEPARTMENT) = 'Math'

This call retrieves information concerning the first occurrence the last name of a student enrolled in the math department. The series of ABDL requests generated for such a call is as follows:

```
[ RETRIEVE  ((TEMPLATE = DEPARTMENT) and
             (DNAME = Math))
            (DNUM) BY DNUM  ]

[ RETRIEVE  ((TEMPLATE =  MAJORING_IN) and
             (DNUM = ****))
            (DATE) BY DATE  ]

[ RETRIEVE  ((TEMPLATE = STUDENT) and
             (DNUM = ****) and
             (DATE = ******))
           (SNUM, LNAME, GRADE) BY SNUM  ]
```

Notice that only the firstt **RETRIEVE** request generated is fully-formed, i.e., may be submitted to MBDS "as-is." Subsequent sequence-field values have been obtained from the execution of previous requests. This process takes place in KC. KMS uses asterisks, as place holders, to mark the maximum allowable length of such sequence fields. Each **RETRIEVE** request, with the exception of the first, is generated solely to extract the functional path to the desired record node. (By doing this, they allow KC to establish and maintain the current position within each record referenced in a DAPLEX call). Consequently, the only attribute in each target list is that of the record sequence field. Of course, the target list of the last request contains all the attributes of the desired node. It is the information obtained from the execution of the final request which is returned to the user, via KFS. Also of note is that each request includes the optional ABDL "BY Attribute_name" clause. The work of Weishar [Ref. : pp. 39-42] has proposed that the results obtained from each **RETRIEVE** request would be sorted by sequence-field value in the language interface. We chose to let KDS (i.e., MBDS) perform this operation

through the inclusion of a "BY sequence_field" clause on all ABDL **RETRIEVE** requests.

2. The DAPLEX INSERT Calls to the ABDL RETRIEVE

This call retrieves information concerning the next occurrence of the last name of a student in the math department, and the majoring in department which have been established as the current DEPARTMENT and MAJORING_IN records within the database by the previous **RETRIEVE** operation (of any type) or **INSERT** operation. If values are to be inserted for each attribute of the record type, there is no requirement to list the attribute names. Only the attribute values need be listed. However, they have to appear in the same order in which they were defined during the original definition of the database. Due to the ABDL requirement that the **INSERT** request include values for all attributes, in the case where the user does not specify values for all attributes in the record, the KMS translator inserts default values. We use a zero (0) and a "z" as the default values for the data types integer and character, respectively.

The DAPLEX **INSERT** consists of attributes, the first of which is preceded by the reserved word **INSERT**. This sequence of attributes has to specify the complete functional path from the parent to the node to be inserted. An example of such a call is as follows:

```
LET NAME(STUDENT) = "Michael Tucker"
LET MINIT(STUDENT) = "A"
INSERT MINIT(NAME(STUDENT))
```

The ABDL request generated for such a call is as follows:

```
[ RETRIEVE  ((TEMPLATE = STUDENT) and
              (SNAME = Michael Tucker))
             (SNUM) BY SNUM   ]

[ INSERT (<TEMPLATE, MINIT>,
         <SMINIT,  A>,
             <FNAME, Michael>,
         <LNAME, Tucker>) ]
```

There is no indication, from the ABDL request generated, that the DAPLEX call contained a looping construct. However, a loop pointer is set during the translation process which allows KC to discern that a looping construct exists and the extent of such a construct. The KMS translator recognizes that the first node of search argument of this DAPLEX call does not specify the parent node as its record type. Consequently, it performs a walk of the functional schema, in reverse order, to obtain the sequence fields required to complete the translation process, i.e., those that specify the complete path from the parent to the record concerned; in this case SNUM.

3. The DAPLEX UPDATE Calls to the ABDL RETRIEVE

The DAPLEX **UPDATE** call is used to retrieve a given record occurrence into a work area, and hold it there so that it may subsequently be updated or deleted. ABDL does not have this requirement. Therefore, when the KMS parser encounters one of these calls, the KMS translator treats them as a corresponding **RETRIEVE** call. Thus the mapping processes described in the previous subsection are applicable to the INSERT calls. Such a call has no meaning with an **INSERT** operator.

4. The DAPLEX DELETE Calls to the ABDL RETRIEVE

The DAPLEX **DELETE** consists of a **UPDATE** call, together with the reserved word **DELETE** immediately following the last attribute in the **UPDATE** portion of the call. When the KMS parser encounters the EXCLUDE portion of the call, the KMS translator generates the appropriate ABDL **RETRIEVE** requests. Then, when the reserved word **DELETE** is parsed, the KMS translator generates appropriate ABDL **DELETE** requests to delete the current record occurrence (i.e., for the current position just established by the **UPDATE** portion of the call), as well as all of the children, grandchildren, etc. (i.e., the descendants) of the current record occurrence. For example, using the example in section

C., subsection 4:

```
DELETE NAME(MAJORING_IN(DEPARTMENT))
WHERE DNAME(DEPARTMENT) = "Math" AND
NAME(PERSON) = "Fred Forest"
```

Assuming that there is only one person named Fred Forest, this call deletes the occurrences of that name within the Math department, but does not delete it from the university's database.

```
[ RETRIEVE  ((TEMPLAE = PERSON) AND
              (PNAME = Fred Forest))
              (SNUM) BY SNUM   ]

[ RETRIEVE  ((TEMPLATE = MAJORING_IN) AND
            (DNUM = ****))
            (DATE) BY DATE   ]

[ DELETE  ((TEMPLATE = MAJORING_IN) AND
          (DNUM = ****))
          (DATE = ******))  ]

[ DELETE  ((TEMPLATE = DEPARTMENT) AND
         (DNUM = ****) AND
         (DATE = ******))  ]

[ DELETE  ((TEMPLATE = PERSON) AND
         (DNAME =  Math))
          (DNUM) BY DNUM  ]

[ DELETE  ((TEMPLATE = STUDENT) AND
          (DNUM = ****) AND
          (DATE = ******))  ]
```

In general , a single **RETRIEVE** request is generated in the **UPDATE** portion of the DAPLEX **DELETE**: (1) if the node type is a child, a single ABDL **DELETE** is generated for that node, and (2) if the node is not a child, a pair of ABDL requests are generated for that node, one **RETRIEVE** and one **DELETE**. Notice that each

RETRIEVE request simple retrieves the sequence-field attribute for the appropriate node ype. The sequence-field values are all that is required, since no information is returned o the user as a result of these **RETRIEVE** requests. These values required to complete he DELETE requests, specifying the complete functional path from the parent to the node o be deleted.

V. THE KERNEL CONTROLLER

The Kernel Controller (KC) is the third module in the DAPLEX language interface and is called by the language interface layer (LIL) when a new database is being created or when an existing database is being manipulated. In either case, LIL first calls the Kernel Mapping System (KMS) which performs the necessary DAPLEX-to-ABDL translations. KC is then called to perform the task of controlling the submission of the ABDL transactions to the multi-backend database system (MBDS) for processing. If the transaction involves inserting, deleting or updating information in an existing database, the control is returned to LIL after MBDS processes the transaction. If the transaction involves a set of retrieval requests, KC sends the translated ABDL requests (which are equivalent to the DAPLEX transaction) to MBDS, receives for the requests the results from MBDS, loads the results into the appropriate file buffer, and displays the results to the user.

These ideas may best be illustrated by examining the following example. Suppose the user issues the following DAPLEX request:

```
FOR EACH Course
  SUCH THAT FOR SOME Offering(Course)
      CNUM(Course) = "MIds" AND
      OFFERING(Date(Course)) = "920326" AND
      GRADE(Student) = "A"
```

This request is translated to the following series of ABDL requests:

```
[ RETRIEVE  ((TEMPLATE = COURSE) AND
             (CTITLE = Mlds))
            (CNUM) BY CNUM   ]

[ RETRIEVE  ((TEMPLATE = OFFERING) AND
             (CNUM = ****) AND
             (DATE = 920326))
             (DATE) BY DATE   ]

[ RETRIEVE  ((TEMPLATE = STUDENT) AND
             (CNUM = ****) AND
             (DATE = ******) AND
             (GRADE = A))
             (SNUM, SNAME, GRADE) BY SNUM   ]
```

KC is now called to control the transmission of these requests to MBDS for execution. Generally, this is accomplished by forwarding the first RETRIEVE request to MBDS. Results are gathered and placed in a file buffer. Notice that the next RETRIEVE is not fully-formed. Therefore, it is necessary to replace the asterisks with a value that is extracted from the first RETRIEVE request in the file buffer. In this example, the value is a course number of CNUM. Again, the request is forwarded to MBDS, and appropriate results are obtained. The last RETRIEVE request is also not fully-formed. In this case, attribute values from both the first and second RETRIEVE requests are utilized to complete the ABDL request. Thus, a value is pulled from the file buffer associated with the second request, and the same number of CNUM is used to form the final request.

The fact that a new value is not pulled from the first request in the file buffer illustrates the currency within the functional database. Specifically, the values that are used in subsequent RETRIEVE requests have to be consistent with those values used in earlier requests. This ensures that the path used to retrieve values from the database is consistent with previous retrievals and the database functions.

The procedures that make up the interface to KDS (therefore, MBDS) are contained in the test interface (TI) of MBDS. To fully integrate KC with KDS, KC calls procedures which are defined in TI.

In this chapter we discuss the processes performed by KC. This discussion is in two parts. First, we examine the data structures relevant to KC, followed by an examination of the functions and procedures found in KC.

A. THE KC DATA STRUCTURES

In this section we review some of the data structures discussed in Chapter II, focusing on those structures that are accessed and used by KC. The first data structure used by KC is the dap_info record shown in Figure 19. The first field, curr_db, is a pointer to the current user of the system; the second field, file, indicates the DAPLEX files requested. The third field, dml_tran, indicates the dml transactions, and the fourth field, ddl_files, indicates to KC where execution of a group of abdl ddl files is to begin. The next three fields are integer fields that contain what DAPLEX operation to be performed (**INSERT, DELETE, RETRIEVE, UPDATE**), the final answer given to the user, or an

error message for the given request. The next field of interest, buff_count, is an integer used to maintain the control of the file buffers associated with the results of each **RETRIEVE** request. For instance, the results associated with the first **RETRIEVE** request of our last example are placed in a file buffer with an extension of "0." The buff_count is incremented by one, and the results associated with the second request are placed in a file buffer with an extension of "1."

```
struct  dap_info
 {
     struct curr_db_infodpi_curr_db;
     struct file_info      dpi_file;
     struct tran_info      dpi_dml_tran;
     struct ddl_info      *dpi_ddl_files;
     int    dap_operation;
     int    dap_answer;
     int    dap_error;
     int    dap_buff_count;
     union  kms_info     dpi_kms_data;
     union  kfs_info     dpi_kfs_data;
     union  kc_info      dpi_kc_data;

 };
```

Figure 16. The dap_info Data Structure.

There is also a structure that supports the tracking of many-to-many relationships in DAPLEX. This structure indicates to KC where the linkage on the group of nodes takes place. The first field is a pointer to a character string. The second field is an integer field for the linkage number, and the final field is a pointer to the next many_to_many node.

```
struct  many_to_many_node

 {
     char   name[ENLength + 1];
     int    link_number;
     struct many_to_many_node*next_m_m;

 };
```

Figure 24. The many_to_many_node Data Structure.

The next data structure is the run_unit structure, which maintains all information about the ancestry of the nodes presently being accessed in the system. The first field is a pointer to the ru_rec_type character string. The second field is also a character string for the ancestor node. Information about the ancestor node is required by the delete and special-retrieve procedures for a proper execution. The next field is a integer that holds the run_unit dbkey request. The last field, is a pointer to the run_unit member.

```
struct   ru_unit
 {
     char   ru_rec_type[RNLength + 1];
     char   ancestor[ANLength + 1];
     int    ru_dbkey;
     struct ru_mem              *ru_memkey;

 };
```

Figure 25. The run_unit Data Structure.

The next structure, ru_mem, is the structure that maintains the member information.

```
struct  ru_mem
 {
   char            *memkey;
   struct ru_mem*next_mem;

 };
```

Figure 26. The ru_mem Data Structure.

The next data structure, cur_record, establishes the current record that is being processed by the system. The first and second fields are pointers to character strings positions in the memory. The next field is an integer field that is the placement holder of the actual current record key. The last structure is a pointer to the next cr_record.

```
struct  cur_record
 {
     char   cr_type[RNLength + 1];
     char   ancestor[ANLength +1];
     int    cr_dbkey;
     struct cur_record*cr_next_rec;

 };
```

Figure 27. The cur_record Data Structure.

The next structure, cur_set, contains the information needed by KC to process a DAPLEX request. The first five fields are pointers for the current set type. This information is utilized by KC to obtain information about the name, ancestor, type, owner and member of the current set. These are all character strings. The next field, own, is a boolean field, and can be true or false. The next field is the current key integer field and the owner key field. The last structure is a pointer to next set of nodes.

```
struct cur_set
  {
      char   cs_set_name[SNLength + 1];
      char   ancestor[ANLength + 1];
      char   cs_type[SNLength + 1];
      char   cs_owner[RNLength + 1];
      char   cs_member[RNLength + 1];
      int    cs_own;
      int    cs_dbkey;
      int    cs_owner_dbkey;
      struct cur_set          *cs_next_set;
```

Figure 28. The cur_set Data Structure.

B. FUNCTIONS AND PROCEDURES

KC makes use of a number of different functions and procedures to manage the transmission of the translated DAPLEX requests (i.e., ABDL requests) to KDS. Not all of these functions and procedures are discussed in detail. Instead, we provide the reader with an overview of how KC controls the submission of the ABDL requests to MDBS.

1. Controlling Requests.

The dap_kc procedure is called whenever LIL has an ABDL transaction for KC o process. This procedure provides the master control over all other procedures used in KC. The first portion of this procedure initialize's global pointers that are used hroughout KC. Specifically, kc_curr_pos is set to point to the first node that is to be processed by KC, and kc_ptr is set to the address of the li_dap structure for a particular user. The remainder of this procedure is a case statement that calls different procedures based upon the type of ABDL transaction being processed. If a new database is being created, the load_tables procedure is called. If the transaction is of any other type, the requests_handler is called. If the transaction is none of the above, there is an error and an error message is generated with the control returned to LIL.

2. Creating a New Database

The creation of a new database is the least difficult transaction that KC handles. The load_tables procedure is called, which performs two functions. First, the test nterface (TI) dbl_template procedure is called. This procedure is used to load the database-template file created by KMS. Next, the TI dbl_dir_tbls procedure is called. This procedure loads the database-descriptor file. These two files represent the attribute-based metadata that is loaded into KDS, i.e., MBDS. After execution of these two procedures, the control returns to LIL.

3. The Insert Request

The INSERT requests are all handled in a similar manner. Suppose the following DAPLEX request is issued by the user:

```
FOR EACH Course
  SUCH THAT FOR SOME Course(Student)
      OFFERING(Date(Course)) = "920326" AND
      GRADE(Student) = "A"
```

This request is translated to the following series of ABDL RETRIEVE requests:

```
[ RETRIEVE  ((TEMPLATE = COURSE) AND
            (CNUM) BY CNUM   ]

[  RETRIEVE  ((TEMPLATE = OFFERING) AND
                (CNUM = ****) AND
                (DATE = 920326))
                (DATE) BY DATE   ]

[  RETRIEVE  ((TEMPLATE = STUDENT) AND
                (CNUM = ****) AND
                (DATE = ******) AND
                (GRADE = A))
                (SNUM, SNAME, GRADE) BY SNUM  ]
```

Also suppose this is the first request the user issues against the database. The kc_curr_pos is set to point to the first ABDL RETRIEVE request shown above. If the kc_curr_pos and cs_set point to the same dap_info node, then KC knows that this is the first request issued by the user, and that the first RETRIEVE is fully-formed .

Since the first RETRIEVE request is complete, it may be immediately forwarded to KDS for execution. This is accomplished by calling run_unit. This procedure uses two TI procedures and the dap_chk_requests_left procedure. In general, the run_unit sends the ABDL request to KDS and waits for the last response to be returned.

After the last response is returned, the control is returned to the cur_record. Now, the cur_rec has to process the remaining **RETRIEVE** requests until the end-of-request flag is detected. Therefore, the kc_curr_pos now points to the kc_curr_pos -> next, i.e, the second **RETRIEVE**. However, this **RETRIEVE** request may not be forwarded to KDS because it is incomplete. Hence, the build_request procedure is called to complete the request. In this instance, a course number (CNUM) is substituted for the place-holding asterisks. This value is obtained from the first **RETRIEVE** in the file buffer. This **RETRIEVE** may now be forwarded to KDS for execution in the same fashion as the first **RETRIEVE**.

4. The Exclude (Delete) Requests

Exclude (Deletes) are the most difficult operations for KC to process. The problem with handling these operations is that they affect the entire database hierarchy as opposed to just a linear path within the database. Suppose the user issues the following DAPLEX request:

```
FOR EACH Course
  SUCH THAT Offering(Course)
  EXCLUDE CNUM(Course) = "Mlds" AND
      OFFERING(Date(Course)) = "920326"
```

This request is translated to the following series of ABDL requests:

```
[ RETRIEVE  ((TEMPLATE = COURSE) AND
          (CTITLE = Mlds))
          (CNUM) BY CNUM   ]

[  RETRIEVE  ((TEMPLATE = OFFERING) AND
           (CNUM = ****) AND
           (DATE = 920326))
           (DATE) BY DATE   ]

[  DELETE  ((TEMPLATE = OFFERING) AND
          (CNUM = ****) AND
          (DATE = ******) ]

[  DELETE  ((TEMPLATE = TEACHER) AND
         (CNUM = ****) AND
         (DATE = ******))  ]

[  DELETE  ((TEMPLATE = STUDENT) AND
         (CNUM = ****) AND
         (DATE = ******))  ]
```

This request first retrieves all course numbers for which the course title is "mlds." This is followed by another **RETRIEVE** request that gathers all dates for a course number

(retrieved above) and an offering data equal to 920326. These **RETRIEVEs** are used to gather the results needed to process the EXCLUDE for this record and its children. The reader may easily discern that we are not only deleting those records for which the course name is "mlds" and the offering date is 920326, but we are also deleting the children of any records for which these conditions are true. Our solution to this problem is the use of the mutual recursion.

VI. CONCLUDING REMARKS

In this thesis, we have concentrated on the model/language interface aspects of using an attribute-based database system, MDBS, as a kernel for the support of the functional data model and the functional query language, DAPLEX. This work is part of the ongoing research being conducted in the Laboratory for Database Systems Research under the direction of Dr. David K. Hsaio. As stated in [Ref. 1], the goal of this phase of the laboratory's research "is to provide increased utility in database computers. A centralized repository of data is made available to multiple, dissimilar hosts. Furthermore, the database is also made available to transactions written in multiple, similar data languages".

The rapid evolution of database technology has provided the motivation for this research. Commercial database management systems have only been available since the 1960's. Today, organizations of all types are critically dependent on the operation of these systems. This dependency comes from the need to centrally control large quantities of operational data. The information must be accurate and readily accessible by relatively inexperienced end-users.

There are three generally known approaches to the design of database systems. These are the network, hierarchical, relational approaches. An organization normally chooses a commercial system based on one of these models. The database must be created and

operator and user personnel must be trained. Because of the re-programming and re-training effort (and money) required, an organization is unlikely to change to a system based on one of the other models.

We have discussed an alternative to the development of separate stand-alone systems for specific data models. In this proposal, the four generally known models and their model-based data languages are supported by the attribute-based data model and data language. We have shown (in the functional case) how a software interface can be built for such support.

Specific contributions of this thesis include extremely thorough explanations of DAPLEX operations, and showing how many of the constructs are directly supportable by ABDL and MDBS. A major design goal has been to design a functional/DAPLEX interface to MDBS without requiring that changes be made to the MDBS system. We have shown that the complete interface can be implemented on a host computer. All translations are accomplished in the functional/DAPLEX interface. MDBS continues to receive and process requests written in the syntax of ABDL. We have also shown that the interface can be designed to utilize existing ABDL constructs (either one or a series of ADBL requests). No changes to the ABDL syntax are required. We have designed the interface to be transparent to the functional/DAPLEX user. The intention is that a functional/DAPLEX user needs to know nothing of the existence of the interface or of MDBS. The user can log in at a system terminal, input a DAPLEX query, and obtain result data in a functional format.

The data modeling capabilities of functional/DAPLEX incorporate those of the hierarchical, relational, and network models. According to Shipman [Ref. 4], the principal characteristics of functional/DAPLEX can be summarized and quoted as follows:

(1) Data is modeled in terms of entities. Database entities are meant to bear a one-to-one correspondence to the "real-world" entities in the user's mental conception of reality.

(2) Relationships between data are expressed as functions, exploiting an established programming metaphor. Identical functional notation is used to reference both "primitive" and "derived" relationships. Conceptual conciseness is enhanced through the use of nested function reference. Functions may be multivalued, returning sets of entities.

(3) The request language is based on the notion of looping through entity sets. Expressions in the language are, in general, set valued. Sets are specified using the functional notation with special operators for qualification and quantification. A simple aggregation semantics, also based on looping, is incorporated. Looping variables are typically declared implicitly.

(4) Computational power is provided through the general-purpose operators of a high-level language. While not emphasized in this paper, this capability is crucial to the development of realistic applications systems.

(5) Derived functions allow users to represent arbitrary entity relationships directly by defining them in terms of existing relationships. In effect, the derived function capability allows application semantics to be encoded into the data description, thereby allowing requests to be expressed directly in terms of those semantics. Updating of derived relationships is supported through procedures explicitly supplied by the user.

(6) Entity types are defined as functions taking no arguments. Notions of subtype and supertype follow naturally from this formulation.

(7) User views are implemented in terms of derived functions.

We have shown that the attribute-based system supports functional/DAPLEX applications. We have provided DAPLEX-to-ABDL translations for selected database operations, and we have proposed a software structure to facilitate implementation, by utilizing MDBS as a kernel database system.

APPENDIX A - THE STRUCTURE DEFINITIONS

```
#define  Overlap        0
#define  Unique         1
#define  New            2
#define  Entity         3
#define  Create         4
#define  Enum           5
#define  String         6
#define  Boolean        7
#define  Integer        8
#define  Float          9
#define  TypeIs         10
#define  SubTypeIs      11
#define  CheckIds       12
#define  NonEnt         13
#define  Derived        14
#define  SubNon         15
#define  GenSub         16

#define  Insert         17
#define  Retrieve       18
#define  LoopParameter  19
#define  Function       20
#define  Assignment     21
#define  Include        22
#define  Exclude        23
#define  Destroy        24
#define  Move           25
#define  Print          26
#define  PrintLine      27

#define  Relation       1
#define  AndRelation    2
#define  OrRelation     3

#define  INTLength      8
#define  FLTLength      16
#define  REQLength      300     /* ABDL request line length */
#define  LITLength      128     /* literal length */
```

```
truct ent_dbid_node
 /* structure def for each entity-relationship dbid node */
{
 char     edn_name[DBNLength + 1];
 struct   ent_non_node   *edn_nonentptr;
 int      edn_num_nonent;  /* number of nonentity types */
 struct   ent_node       *edn_entptr;
 int      edn_num_ent;      /* number of entity types */
 struct   gen_sub_node   *edn_subptr;
 int      edn_num_gen;      /* number of gen_subtypes */
 struct   sub_non_node   *edn_nonsubptr;
 int      edn_num_nonsub;  /* number of nonentity subtypes */
 struct   der_non_node   *edn_nonderptr;
 int      edn_num_der;      /* number or nonentity derived types */
 struct   overlap_node   *edn_ovrptr;
 int      edn_num_ovr;      /* number of overlap_nodes */
 struct   ent_dbid_node  *edn_next_db;
};



truct ent_node
 /* structure definition for each entity node */
{
 char     en_name[ENLength + 1];
 int      en_last_ent_id;       /* keeps track of the unique id
                assigned to each entity type in the database */
 int      en_num_funct;         /* number of assoc. functions  */
 int      en_terminal;      /* if true (=1) it is a terminal type */
 struct   function_node  *en_finptr;
 struct   ent_node       *en_next_ent;
};




truct gen_sub_node
 /* structure def for each generalization (supertype/subtype) node */
{
 char     gsn_name[ENLength + 1];
 int      gsn_num_funct;    /* number of assoc. functions*/
 int      gsn_terminal;     /* if true (=1) it is a terminal type */
 struct   ent_node_list  *gsn_entptr; /* ptr to entity supertype */
 int      gsn_num_ent;      /* number of entity supertypes */
 struct   function_node  *gsn_finptr;
 struct   sub_node_list  *gsn_subptr; /* ptr to subtype supertype */
 int      gsn_num_sub;      /* number of subtype supertypes */
 struct   gen_sub_node      *gsn_next_genptr;
};
```

```
struct ent_non_node
  /* structure def for each base-type nonentity node */
 {
  char     enn_name[ENLength + 1];
  char     enn_type;                  /* either i(nteger), s(tring),
                                  f(loat), e(numeration), or b(oolean) */
  int      enn_total_length;          /* max length of base-type value */
  int      enn_range;                  /* true or false depending
                                       on whether there is a
                                       range.  If range exists,
                                       there must be two entries
                                       into ent_value */
  int      enn_num_values;               /* number of actual values */
  struct   ent_value      *enn_value;  /* actual value of base-type */
  int      enn_constant;          /* boolean to refelect constant value */
  struct   ent_non_node    *enn_next_node;
 };


struct sub_non_node
  /* structure def for each subtype nonentity node */
 {
  char     snn_name[ENLength + 1];
  char     snn_type;                  /* either i(nteger), s(tring),
                                  f(loat), e(numeration), or b(oolean) */
  int      snn_total_length;          /* max length of subtype value */
  int      snn_range;                  /* true or false depending
                                       on whether there is a
                                       range.  If range exists,
                                       there must be two entries
                                       into ent_value */
  int      snn_num_values;               /* number of actual values */
  struct   ent_value      *snn_value;  /* actual value of subtype */
  struct   sub_non_node    *snn_next_node;
 };


struct der_non_node
  /* structure def for each derived type nonentity node */
 {
  char     dnn_name[ENLength + 1];
  char     dnn_type;              /* either i(nteger), s(tring),
                                 f(loat), e(numeration), or b(oolean) */
  int      dnn_total_length;      /* max length of derived type value */
  int      dnn_range;             /* true or false depending on whether
                                        there is a range.  If range exists,
                                        there must be two entries in the
                                        ent_value */
  int      dnn_num_values;   /* number of actual values */
  struct   ent_value      *dnn_value; /* actual value of derived type */
  struct   der_non_node    *dnn_next_node;
 };
```

```
struct overlap_node
  /* structure def for overlapping constraint */
{
  char   base_type_name[ENLength+1];
  struct sub_node_list          *snlptr;
  int    num_sub_node;              /* number of sub_node in the above list */
  struct overlap_node           *next;
};



struct function_node
  /* structure definition for each function type declaration */
{
  char    fn_name[ENLength+1];
  char    fn_type;              /* either f(loat), i(nteger), s(tring),
                                   b(oolean), or e(numeration) */
  int     fn_range;             /* Boolean if range of values */
  int     fn_set;               /* Boolean if set of values */
  int     fn_total_length;      /* max length */
  int     fn_num_value;                 /* number of actual values */
  struct  ent_value    *fn_value;  /* actual value */
  struct  ent_node     *fn_entptr; /* ptr to entity type */
  struct  gen_sub_node *fn_subptr; /* ptr to entity subtype */
  struct  ent_non_node *fn_nonentptr; /* ptr to nonentity type */
  struct  sub_non_node *fn_nonsubptr; /* ptr to nonentity subtype */
  struct  der_non_node *fn_nonderptr; /* ptr to nonentity dertype */
  int     fn_entnull;             /* initialized false set true for no
                                     value */
  int     fn_unique;              /* init false - unique if true */
  struct  function_node *next;
};



struct sub_node_list      /* list of ponters */
  /* structure definition for terminal subtypes that define one or more
     subtypes */
{
  struct  gen_sub_node    *subptr;  /* only terminal subtypes */
  struct  sub_node_list   *next;
};



struct ent_node_list      /* list of pointers */
  /* structure definition for subtypes with one or more entity supertypes */
{
  struct  ent_node        *entptr;
  struct  ent_node_list   *next;
};
```

```
struct ent_value
  /* struct def for value of 'i','s','f','e', or 'b' */
 {
  char    *ev_value;     /* pointer to character string only */
  struct  ent_value      *next;
 };



struct ident_list
 {
  char   name [ENLength + 1];
  struct ident_list      *next;
 };



struct ent_value_list
 {
  char   type;
  int    num_values;
  struct ent_value              *ev_ptr;
 };



struct   dap_kms_info
 {
  struct  ident_list            *dki_temp_ptr;
  struct  ident_list            *dki_name1_ptr;
  struct  ident_list            *dki_id_ptr;
  struct  sub_node_list         *dki_overfirst_ptr;
  struct  der_non_node          dki_der_non;
  struct  sub_non_node          dki_sub_non;
  struct  ent_non_node          dki_ent_non;
  struct  function_node         dki_funct;
  struct  ent_value             *dki_ev_ptr;
  struct  dap_create_list       *dki_create;
  struct  req_line_list         *dki_req_ptr;
  struct  create_ent_list       *dki_cel_ptr;
  struct  overlap_node          *dki_create_ovrptr;
  struct  ent_value_list                        dki_evl_ptr;
  struct  dml_statement         *dml_statement_ptr;
  struct  loop_info             *loop_info_ptr;
 };



struct dap_create_list
 {
  int     req_type;             /* Insert or Retrieve */
  char     en_name[ENLength + 1];
  struct  dap_av_pair_list      *av_pair_ptr;
  struct  dap_create_list       *next;
 };
```

```
struct dap_av_pair_list
{
  char    name[ENLength + 1];
  struct  function_node       *fInptr;
  int     num_value;
  struct  ent_value           *valptr;
  struct  dap_av_pair_list    *next;
};



struct req_line_list
{
  char    req_line[REQLength];
  struct  req_line_list *next;
};



struct create_ent_list
{
  struct ent_node_list                *enl_ptr;
  struct sub_node_list                *snl_ptr;
  struct create_ent_list      *next;
};



struct dap_expr_info
{
  int    relation_type;      /* Relation, AndRelation or OrRelation */
  struct relation_list           *rel_list_ptr;
};









struct relation_list
{
  struct simple_expr1                 *simple_expr1_ptr;
  struct simple_expr2                 *simple_expr2_ptr;
  struct simple_expr3                 *simple_expr3_ptr;
  struct simple_expr4                 *simple_expr4_ptr;
  struct relation_list        *next;
};
```

```
struct simple_expr1
{
  char   lit_array[LITLength+1];
  struct set_constructor        *set_construct_ptr;
  struct indexed_component              *indexed_comp_ptr;
  struct funct_appln            *funct_appln_ptr;
};



struct funct_appln
{
  int    type;                  /* COUNT, SUM, AVG, MIN, MAX */
  /* The followings are structures for expr_types in the grammar */
  char          name_id[ENLength + 1];
  struct set_constructor        *set_construct_ptr;
  struct indexed_component   *indexed_comp_ptr;
};



struct indexed_component
{
  char   name_id[ENLength + 1];
  char   type;      /* Entity, GenSub, LoopParameter, Function */
  char   parent_name[ENLength + 1];
  struct indexed_component   *next;
};


struct set_constructor
{
  struct basic_expr_list        *basic_expr_ptr;
  struct set_construct2         *set_construct2_ptr;
  struct set_construct3         *set_construct3_ptr;
};



struct set_construct2
{
  struct simple_expr1                   *simple_expr1_ptr;
  char   name1[ENLength + 1];
  struct dap_expr_info                  dap_expr_ptr;
};



struct dap_range_info
{
  int    range_type;            /* Integer or Float */
  char   first_value[FLTLength + 1];
  char   second_value[FLTLength + 1];
};
```

```
struct set_construct3
 {
  struct simple_expr1                *simple_expr1_ptr;
  struct dap_range_info      dap_range;
  struct dap_expr_info               dap_expr_ptr;
 };



struct basic_expr_list
 {
  char   lit_array[LITLength+1];
  struct indexed_component   *indexed_comp_ptr;
  struct funct_appln         *funct_appln_ptr;
  struct basic_expr_list     *next;
 };


struct comp_assoc_list
 {
  char     name[ENLength + 1];
  struct   simple_expr1       simple_expr;
  struct   comp_assoc_list    *next;
 };



struct simple_expr2
 {
  struct simple_expr1        *first_expr;
  int    rel_operator;
  struct simple_expr1        *second_expr;
 };



struct simple_expr3
 {
  struct simple_expr1        *simple_expr;
  int    in_op;              /* INOp or NINOp */
  struct dap_range_info      *dap_range;
 };



struct simple_expr4
 {
  struct simple_expr1                *simple_expr;
  int    in_op;                      /* INOp or NINOp */
  char   name_id[ENLength + 1];
 };
```

```
struct domain_info
 {
   /* only one of the first two fields is active at one time */
   /* see loop_expr in grammar */
   struct indexed_component   *indexed_comp_ptr;
   char   name[ENLength + 1];
   struct dap_expr_info                *dap_expr_ptr;   /* optional field */
 };



struct loop_info
 {
   char   loop_parameter[ENLength + 1];
   struct ent_node              *entptr;
   struct gen_sub_node                  *subptr;
   struct domain_info           domain;
   struct order_comp_list       *order_comp_ptr;
   struct dml_statement2_list   *dml_statement2_list_ptr;
 };



struct order_comp_list
 {
   int sort_order;              /* ASCENDING or DESCENDING */
   struct indexed_component     *indexed_comp_ptr;
   struct order_comp_list       *next;
 };



struct dml_statement2_list
 {
   struct dml_statement                 *dml_statement2_ptr;
   struct dml_statement2_list   *next;
 };



struct dml_statement
 {
   int    type;          /*  Assignment, Include, Exclude,
                                        Destroy, Move, Procedure, Create */
   struct dap_expr_info         *dap_expr_ptr;
   struct indexed_component     *indexed_comp_ptr;
   struct basic_expr_list       *basic_expr_ptr;
   struct comp_assoc_list       *comp_assoc_ptr;
 };
```

APPENDIX B. THE LIL SPECIFICATIONS

```
module DAPLEX-INTERFACE

f_build_ddl_files()
  /* This routine is used to create the MBDS template and descriptor files */
{
  struct   ddl_info   *ddl_info_alloc();

#ifdef EnExFlag
    printf("Enter f_build_ddl_file\n");
#endif

  if (dap_info_ptr->dpi_ddl_files == NULL)
     dap_info_ptr->dpi_ddl_files = ddl_info_alloc();

  f_build_template_file();
  f_build_desc_file();

#ifdef EnExFlag
    printf("Exit f_build_ddl_file\n");
#endif

}


f_build_template_file()
{
  /* This routine builds the MBDS template file for a new daplex  */
  /* database that was just created.  */

  struct ent_dbid_node    *db_ptr;
  struct ent_node         *ent_ptr;
  struct gen_sub_node     *gen_ptr;
  struct function_node    *funct_ptr;
  struct file_info        *f_ptr;
  char                    temp_str[NUMDIGIT + 1];
  char                        get_fun_type();
  int                     i;

#ifdef EnExFlag
    printf("Enter f_build_template_file\n");
#endif
```

```
   /* Begin by setting the pointers to the dap_info data structure */
   /* that is maintained for each user of the system.              */
   db_ptr = dap_info_ptr->dpi_curr_db.cdi_db.dn_fun;
   f_ptr  = &(dap_info_ptr->dpi_ddl_files->ddli_temp);

   /* Next, copy the filename where the MBDS template information will */
   /* be stored.  This filename is constant and was obtained from      */
   /* licommdata.def.  */
   strcpy(f_ptr->fi_fname, FTEMPFname);

   /*  Next, open the template File to be created:                     */
   if ((f_ptr->fi_fid = fopen(f_ptr->fi_fname,"w")) == NULL)
     {
     printf("Unable to open %s\n", FTEMPFname);

#ifdef EnExFlag
     printf("Exit1 f_build_template_file\n");
#endif
     return;
     };

   /*  Next, write out the database name & number of files :           */
   fprintf(f_ptr->fi_fid, "%s\n", db_ptr->edn_name);
   num_to_str(db_ptr->edn_num_ent + db_ptr->edn_num_gen, temp_str);
   fprintf(f_ptr->fi_fid, "%s\n", temp_str);

   /*  Next, set the pointer to the first entity:                      */
   ent_ptr = db_ptr->edn_entptr;

   /*  While there are more entities to process, write out the number  */
   /*  of functions (+2 for the attribute "TEMP" and the key value attribute) */
   /*  and the entity name:       */
   while (ent_ptr)
     {
     num_to_str((ent_ptr->en_num_funct + 2), temp_str);
     fprintf(f_ptr->fi_fid, "%s\n", temp_str);
     fprintf(f_ptr->fi_fid, "%s\n", ent_ptr->en_name);

     /*  Print out the constant attribute "TEMP s" and key value attribute   */
     fprintf(f_ptr->fi_fid, "TEMP s\n");
     fprintf(f_ptr->fi_fid, "%s i\n", ent_ptr->en_name);

     /*  Now, set the pointer to the first function:                     */
     funct_ptr = ent_ptr->en_ftnptr;
     wr_all_funct_attr(f_ptr->fi_fid, funct_ptr);
```

```
    /*  set the pointer to the next entity:                          */
    ent_ptr = ent_ptr->en_next_ent;

    }   /* end while ent_ptr */

  /*  Next, set the pointer to the first gen sub node :              */
  gen_ptr = db_ptr->edn_subptr;

  /*  While there are more sub nodes to process, write out the number    */
  /*  of functions (+2 for the attribute "TEMP" and the key value attribute) */
  /*  and the entity name:       */
  while (gen_ptr)
     {
    num_to_str((gen_ptr->gsn_num_funct + 2), temp_str);
    fprintf(f_ptr->fi_fid, "%s\n", temp_str);
    fprintf(f_ptr->fi_fid, "%s\n", gen_ptr->gsn_name);

    /*  Print out the constant attribute "TEMP s" and key value attribute   */
    fprintf(f_ptr->fi_fid, "TEMP s\n");
    fprintf(f_ptr->fi_fid, "%s i\n", gen_ptr->gsn_name);

    /*  Now, set the pointer to the first function:                  */
    funct_ptr = gen_ptr->gsn_ftnptr;
    wr_all_funct_attr(f_ptr->fi_fid, funct_ptr);

    /*  set the pointer to the next gen sub node:                    */
    gen_ptr = gen_ptr->gsn_next_genptr;

    }   /* end while gen_ptr */

  /*  Finally, close out the file and exit this routine:             */
  fclose(f_ptr->fi_fid);

#ifdef EnExFlag
    printf("Exit2 f_build_template_file\n");
#endif
}


wr_all_funct_attr(fid, funct_ptr)
  FILE    *fid;
  struct  function_node    *funct_ptr;
  {

#ifdef EnExFlag
    printf("Enter wr_all_funct_attr\n");
#endif
```

```
    /*  While there are more attributes to process,      */
    /*  print out attr. name & type:        */
    while (funct_ptr)
       {
       fprintf(fid, "%s %c\n", funct_ptr->fn_name, get_fun_type(funct_ptr));

       /*  Set the pointer to the next function:                    */
       funct_ptr = funct_ptr->next;

       }  /* end while funct_ptr */

#ifdef EnExFlag
    printf("Exit wr_all_funct_attr\n");
#endif
}

char get_fun_type(fptr)
    struct  function_node        *fptr;
{
            char    fun_type;

#ifdef EnExFlag
    printf("Enter get_fun_type\n");
#endif

     switch (fptr->fn_type)
       {
           case 'i' : ;
           case 'f' : ;
           case 's' : fun_type = fptr->fn_type;
                      break;
           case 'b' : fun_type = 'i';
                      break;
           case 'e' : if (fptr->fn_entptr ||
                          fptr->fn_subptr)
                          fun_type = 'i';
                      else if (fptr->fn_nonentptr)
                          fun_type = fptr->fn_nonentptr->enn_type;
                      else if (fptr->fn_nonsubptr)
                          fun_type = fptr->fn_nonsubptr->snn_type;
                      else if (fptr->fn_nonderptr)
                          fun_type = fptr->fn_nonderptr->dnn_type;
                      if (fun_type =='e')/* still not i or f type */
                          fun_type = 's';
       }  /* end switch */
```

```
#ifdef EnExFlag
    printf("Exit get_fun_type\n");
#endif
    return(fun_type);

} /* end get_fun_type */



/* builddesc.c */



_build_desc_file()
{
  /* This routine builds the Descriptor File to be used by the MBDS in the   */
  /* creation of indexing clusters:                                          */

  struct    ent_dbid_node     *db_ptr;          /*  database pointer       */
  struct    ent_node          *ent_ptr;         /*  entity node ptr        */
  struct    gen_sub_node      *gen_ptr;         /*  gen sub node ptr       */
  struct    descriptor_node   *desc_head_ptr,   /*  pointers to Desc_node..*/
                              *ask_all_fun_nodes();
  struct    file_info         *f_ptr;           /*  File pointer           */
  int                         index,
                              str_len;

#ifdef EnExFlag
    printf("Enter f_build_desc_file\n");
#endif

  /*  Begin by setting the pointers to the dap_info data structure that is   */
  /*  maintained for each user of the system:                                */
  db_ptr = dap_info_ptr->dpi_curr_db.cdi_db.dn_fun;
  f_ptr = &(dap_info_ptr->dpi_ddl_files->ddli_desc);

  /*  Next, copy the filename where the MBDS Descriptor File information     */
  /*  will be stored.  This filename is Constant, and was obtained from      */
  /*  licommdata.def:                                                        */
  strcpy(f_ptr->fi_fname, FDESCFname);

  /*  Now, open the Descriptor File to be created:                           */
  f_ptr->fi_fid = fopen(f_ptr->fi_fname, "w");
```

```
/*  The next step is to traverse the Linked List of entities in the data-  */
/*  base.  There are two reasons for doing so:  First, to write the        */
/*  entity Names to the Descriptor File as EQUALITY Descriptors; this is   */
/*  done automatically with any Daplex Database, is a necessary element    */
/*  of any Descriptor File created from such a Database, and requires      */
/*  no user involvement.  Second, it allows us to present the Entity       */
/*  Names (without their respective Attributes) to the User, as a memory   */
/*  jog:                                                                   */

system("clear");
fprintf(f_ptr->fi_fid, "%s\n", db_ptr->edn_name);
fprintf(f_ptr->fi_fid, "TEMP C s\n");
printf("\nThe following are the Entities in the ");
printf("%s Database:\n\n", db_ptr->edn_name);

ent_ptr = db_ptr->edn_entptr;

/*  Traverse all entity nodes                  */
while (ent_ptr)
   {
   fprintf(f_ptr->fi_fid, "! %c", ent_ptr->en_name[0] );
   str_len = strlen( ent_ptr->en_name );
   for(index = 1; index < str_len; index++)
      if (isupper(ent_ptr->en_name[index]))
          fprintf(f_ptr->fi_fid, "%c", tolower( ent_ptr->en_name[index] ));
      else
          fprintf(f_ptr->fi_fid, "%c", ent_ptr->en_name[index]);
   fprintf(f_ptr->fi_fid, "\n");
   printf("\n\t%s", ent_ptr->en_name);
   ent_ptr = ent_ptr->en_next_ent;
   }   /*  End "while (ent_ptr)"  */

gen_ptr = db_ptr->edn_subptr;

/*  Traverse all gen sub nodes                 */
while (gen_ptr)
   {
   fprintf(f_ptr->fi_fid, "! %c", gen_ptr->gsn_name[0] );
   str_len = strlen( gen_ptr->gsn_name );
   for(index = 1; index < str_len; index++)
      if (isupper(gen_ptr->gsn_name[index]))
          fprintf(f_ptr->fi_fid, "%c", tolower( gen_ptr->gsn_name[index] ));
      else
          fprintf(f_ptr->fi_fid, "%c", gen_ptr->gsn_name[index]);
   fprintf(f_ptr->fi_fid, "\n");
   printf("\n\t%s", gen_ptr->gsn_name);
   gen_ptr = gen_ptr->gsn_next_genptr;
   }   /*  End "while (gen_ptr)"  */
```

```
/*  Each Descriptor Block must be followed by the "@" sign:              */
fprintf(f_ptr->fi_fid, "@\n");

/*  Now, inform the user of the procedure that must be followed to create  */
/*  the Descriptor File:                                                   */
printf("\n\nBeginning with the first Entity, we will present each");
printf("\nfunction of the entity.  You will be prompted as to whether");
printf("\nyou wish to include that function as an Indexing Attribute,");
printf("\nand, if so, whether it is to be indexed based on strict");
printf("\nEQUALITY, or based on a RANGE OF VALUES. If you do not want");
printf("\nto enter any indexes for your database, type an 'n' when");
printf("\nthe Action --> prompt appears");
printf("\n\nStrike RETURN or 'n' when ready to continue.");
dap_info_ptr->dap_answer = get_ans(&index);

/*  Initialize the pointer to a Linked List that will hold the results     */
/*  of the Descriptor Values, then return to the first entity of the       */
/*  database and begin cycling through the individual attributes:          */
desc_head_ptr = NULL;

ent_ptr = db_ptr->edn_entptr;
while ((ent_ptr) && (dap_info_ptr->dap_answer != 'n'))
   {
   desc_head_ptr =
    ask_all_fun_nodes(desc_head_ptr, ent_ptr->en_name, ent_ptr->en_ftnptr);

   ent_ptr = ent_ptr->en_next_ent;
   }  /*  End while */

gen_ptr = db_ptr->edn_subptr;
while ((gen_ptr) && (dap_info_ptr->dap_answer != 'n'))
   {
   desc_head_ptr =
    ask_all_fun_nodes(desc_head_ptr, gen_ptr->gsn_name, gen_ptr->gsn_ftnptr);
   gen_ptr = gen_ptr->gsn_next_genptr;
   }  /*  End while */

/*  Now, we will traverse the Linked List of Descriptor Attributes and     */
/*  Values which was created, writing them to our Descriptor File:         */

   wr_all_desc_values(f_ptr->fi_fid, desc_head_ptr);
   fclose(f_ptr->fi_fid);

#ifdef EnExFlag
   printf("Exit f_build_desc_file\n");
#endif
  }
```

```
struct descriptor_node *ask_all_fun_nodes(desc_head_ptr, en_name, funct_ptr)
  struct  descriptor_node   *desc_head_ptr;
  char    en_name[ENLength + 1];
  struct  function_node     *funct_ptr;
  {
  struct  descriptor_node   *desc_node_ptr,
                            *descriptor_node_alloc(); /* Allocates Nodes */
  struct   value_node       *valuenode_ptr;     /* points to Value Node  */
  int                       num,              /*                       */
                            found,            /* Boolean flag          */
                            goodanswer;       /* Boolean flag          */

#ifdef EnExFlag
    printf("Enter ask_all_fun_nodes\n");
#endif

  while (funct_ptr)
   {
   if (funct_ptr->fn_entptr == NULL && funct_ptr->fn_subptr == NULL)
     {

     system("clear");
     printf("Entity name:  %s\n", en_name);
     printf("Function Name: %s\n\n", funct_ptr->fn_name);

     /*  Now, traverse the Attribute linked list that is being created,  */
     /*  to see if the current Attribute has already been established as */
     /*  a Descriptor Attribute.  If so, offer the user the opportunity  */
     /*  to add additional EQUALITY or RANGE OF VALUE values; otherwise, */
     /*  offer the user the opportunity to establish this as a Descriptor */
     /*  Attribute:                                                      */
     desc_node_ptr = desc_head_ptr;
     found = FALSE;
     while ((desc_node_ptr) && (found == FALSE))
       {
       if (strcmp(funct_ptr->fn_name, desc_node_ptr->attr_name) == 0)
         {

         /*  The Attribute HAS already been chosen as a Descriptor.     */
         /*  Allow the user the option of adding additional Descriptor  */
         /*  values, after listing those already entered:               */

         printf("\nThis Attribute has been chosen as an Indexing Attribute.\n");
         printf("The following are the values that have been specified:\n\n");
         found = TRUE;
         valuenode_ptr = desc_node_ptr->first_value_node;
```

```
    while (valuenode_ptr)
      {
      if (desc_node_ptr->descriptor_type == 'A')
        printf("\t%s %s\n", valuenode_ptr->value1,
                        valuenode_ptr->value2);
      else
        printf("\t%s\n", valuenode_ptr->value2);
      valuenode_ptr = valuenode_ptr->next_value_node;
      }  /*  End "while (valuenode_ptr)"  */
    printf("\nDo you wish to add more ");
    if (desc_node_ptr->descriptor_type == 'A')
      printf("RANGE");
    else
      printf("EQUALITY");
    printf(" values?  (y or n)\n");
    dap_info_ptr->dap_answer = get_ans(&num);
    if ((dap_info_ptr->dap_answer == 'y') ||
        (dap_info_ptr->dap_answer == 'Y'))

      /*  The user DOES wish to add more descriptors to the       */
      /*  currently existing list:                                */
      {
      if (desc_node_ptr->descriptor_type == 'A')
        build_RAN_descrip(desc_node_ptr, funct_ptr->fn_total_length);
      else
        build_EQ_descrip(desc_node_ptr, funct_ptr->fn_total_length);
      }  /*  End "if ((dap_info_ptr->dap_answer == 'y') ||
                  (dap_info_ptr->dap_answer == 'Y'))         */
   }  /*  End "if (strcmp(...) == 0)"  */
  desc_node_ptr = desc_node_ptr->next_desc_node;
  }  /*  End "while ((desc_node_ptr) && (found..))"  */

if (found == FALSE)

  /*  The Attribute has NOT previously been chosen as a Descriptor. */
  /*  Allow the user the option of making this a Descriptor Attri-  */
  /*  bute, with appropriate Descriptor Values:                     */
  {
  printf("\nDo you wish to install this function as an ");
  printf("Indexing Attribute?\n\n");
  printf("\t(n) - no; continue with next Attribute/Relation\n");
  printf("\t(e) - yes; establish this as an EQUALITY Attribute\n");
  printf("\t(r) - yes; establish this as a RANGE Attribute\n");
  goodanswer = FALSE;
  while (goodanswer == FALSE)
    {
    dap_info_ptr->dap_answer = get_ans(&num);
```

```
    switch(dap_info_ptr->dap_answer)
      {
      case 'n':  /*  User does NOT want to use this as an        */
                 /*  Indexing (Descriptor) Attribute:            */
                 goodanswer = TRUE;
                 break;

      case 'e':  /*  User wants to use this as an EQUALITY       */
                 /*  Attribute:                                  */
                 goodanswer = TRUE;
                 desc_node_ptr = descriptor_node_alloc();
                 desc_node_ptr->next_desc_node = desc_head_ptr;
                 desc_head_ptr = desc_node_ptr;
                 strcpy(desc_node_ptr->attr_name, funct_ptr->fn_name);
                 desc_node_ptr->descriptor_type = 'B';
                 desc_node_ptr->value_type = get_fun_type(funct_ptr);
                desc_node_ptr->first_value_node = NULL;
                build_EQ_descrip(desc_node_ptr,
                                 funct_ptr->fn_total_length);
                 break;

      case 'r':  /*  User wants to use this as a RANGE Attribute: */
                 goodanswer = TRUE;
                 desc_node_ptr = descriptor_node_alloc();
                 desc_node_ptr->next_desc_node = desc_head_ptr;
                 desc_head_ptr = desc_node_ptr;
                 strcpy(desc_node_ptr->attr_name, funct_ptr->fn_name);
                 desc_node_ptr->descriptor_type = 'A';
                 desc_node_ptr->value_type = get_fun_type(funct_ptr);
                desc_node_ptr->first_value_node = NULL;
                 build_RAN_descrip(desc_node_ptr,
                                   funct_ptr->fn_total_length);
                 break;

      default:   /*  User did not select a valid choice:         */
                 printf("\nError - Invalid operation selected;\n");
                 printf("Please pick again\n");
                 break;

      }  /*  End Switch  */

    }  /*  End "While (goodanswer = FALSE)"  */

  }  /*  End "if (found == FALSE)"  */

}  /* End "if (funct_ptr->fn_entptr ...." */
```

```
    funct_ptr = funct_ptr->next;
   }  /*  End "while (funct_ptr)"  */

#ifdef EnExFlag
    printf("Exit ask_all_fun_nodes\n");
#endif

   return(desc_head_ptr);

}

wr_all_desc_values(fid, desc_head_ptr)

/* This routine traverse the linked list of descriptor attributes      */
/* and write all descriptor values out to the Descriptor file          */

  FILE   *fid;
  struct descriptor_node      *desc_head_ptr;
  {
  struct descriptor_node      *desc_node_ptr;
  struct value_node           *valuenode_ptr;     /* points to Value Node   */

#ifdef EnExFlag
    printf("Enter wr_all_desc_values\n");
#endif

  desc_node_ptr = desc_head_ptr;

  while (desc_node_ptr)
     {
     if (desc_node_ptr->first_value_node)
        {
        fprintf(fid, "%s %c %c\n", desc_node_ptr->attr_name,
                                   desc_node_ptr->descriptor_type,
                                   desc_node_ptr->value_type);
        valuenode_ptr = desc_node_ptr->first_value_node;
        while (valuenode_ptr)
           {
           fprintf(fid,"%s %s\n", valuenode_ptr->value1,
                                  valuenode_ptr->value2);
           valuenode_ptr = valuenode_ptr->next_value_node;
           }  /*  End "while (valuenode_ptr)"  */

        fprintf(fid,"@\n");
        }  /*  End "if (desc_node_ptr->first_value_node)  */
```

```
      desc_node_ptr = desc_node_ptr->next_desc_node;

      }  /*  End "while (desc_node_ptr)"  */

   fprintf(fid, "$\n");

#ifdef EnExFlag
     printf("Exit wr_all_desc_values\n");
#endif
}
```

LIST OF REFERENCES

1. Demurjian, S. A. and Hsiao, D. K., "New Directions in Database-Systems Research and Development," in the Proceedings of the Directions in Computing Conference, Trondheim, Norway, August, 1985; also in Technical Report, NPS-52-85-001, Naval Postgraduate School, Monterey, California, February 1985.

2. Banerjee, J., Hsiao, D. K., and Ng, F., "Database Transformation, Query Translation and Performance Analysis of a Database Computer in Supporting Hierarchical Database Management," IEEE Transactions on Software Engineering, March 1980.

3. Hsiao, D. K., and Harary, F., "A Formal System for Information Retrieval from Files," Communications of the ACM, Vol. 13, No. 2, February 1970, also in Corrigenda, Vol 13., No. 4, April 1970.

4. Wong, E., and Chang, T. C., "Canonical Structure in Attribute Based File Organization," Communications of the ACM, September 1971.

5. Shipman, D. W., "The Functional Data Model and the Data Language DAPLEX,"ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981.

6. Date, C. J. An Introduction to Database Systems, 3d ed., Addison Wesley, 1982.

7. Elmasri, R. and Navathe, S. B. "Advanced Data Modeling Concepts," in the Fundamentals of Database Systems. The Benjamin/Cummings Publishing Company, Inc., 1989.

8. The Ohio State University, Columbus, Ohio, Technical Report No. OSU-CISRC-TR-77-7, DBC Software Requirements for Supporting Relational Databases, by J. Banerjee and D. K. Hsiao, November 1977.

9. Boehm, B. W., Software Engineering Economics, Prentice-Hall, 1981.

10. Naval Postgraduate School, Monterey, California, Technical Report, NPS52-84-012, Software Engineering Techniques for Large-scale Database Systems as Applied to the Implementation of a Multi-Backend System, by Ali Orooji, Douglas Kerr and David K. Hsiao, August 1984.

11. Johnson, S. C., YACC: Yet Another Compiler-Compiler, Bell Laboratories, Murray Hill, New Jersey, July 1978.

12. Kernighan, B. W. and Ritchie, D. M., The C Programming Language, Prentice-Hall, 1978.

13. Lesk, M. E. and Schmidt, E., Lex - A Lexical Analyzer Generator, Bell Laboratories, Murray Hill, New Jersey, July 1978.

14. Kloepping, G. R., and Mack, J. F., The Design and Implementation of a Relational Interface for the Multi-Lingual Database System. M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

15. Benson, T. P. and Wentz, G. L., The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System. M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

16. Wortherly, C. R., The Design and Implementation of a Network Interface for the Multi-Lingual Database System. M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943-5100	2
Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	2
Professor David K. Hsiao, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 939-5100	2
Professor Magda Kamel, Code AS Administrative Sciences Naval Postgraduate School Monterey, CA 93943-5100	1
Sybil B. Mack 4418 Velpoe Drive Columbus, GA 31907	3